

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF

关于深度学习的专题性著作，也是掌握卷积神经网络内涵的进阶之书

Deep Learning
Mastering Convolutional Neural Networks from Beginner

深度学习

卷积神经网络从入门到精通

李玉鑑 张婷 单传辉 刘兆英 等著



机械工业出版社
China Machine Press



主要作者介绍

李玉鑑（鉴）

北京工业大学教授，博士生导师。华中理工大学（现名为华中科技大学）本科毕业，中国科学院数学研究所硕士毕业，中国科学院半导体研究所博士毕业，北京邮电大学博士后出站。曾在中国科学院生物物理所工作，对意识的本质问题关注过多年，并在《21世纪100个交叉科学难题》上发表《揭开意识的奥秘》一文，提出了解决意识问题的认知相对论纲领，对脑计划和类脑研究具有宏观指导意义。长期围绕人工智能的核心目标，在神经网络、自然语言处理、模式识别和机器学习等领域开展教学、科研工作，发表国内外期刊、会议论文数十篇，是本书和《深度学习导论及案例分析》的第一作者。



智能系统与技术丛书

Deep Learning
Mastering Convolutional Neural Networks from Beginner

深度学习

卷积神经网络从入门到精通

李玉鑑 张婷 单传辉 刘兆英 等著



机械工业出版社
China Machine Press



图书在版编目 (CIP) 数据

深度学习: 卷积神经网络从入门到精通 / 李玉鑑等著. —北京: 机械工业出版社, 2018.7
(智能系统与技术丛书)

ISBN 978-7-111-60279-8

I. 深… II. 李… III. 学习系统 IV. TP273

中国版本图书馆 CIP 数据核字 (2018) 第 137837 号

深度学习: 卷积神经网络从入门到精通

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 张梦玲

责任校对: 李秋荣

印刷: 北京市兆成印刷有限责任公司

版次: 2018 年 7 月第 1 版第 1 次印刷

开本: 186mm×240mm 1/16

印张: 27

书号: ISBN 978-7-111-60279-8

定价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东



P R E F A C E

前 言

随着谷歌的 AlphaGo、IBM 的 Watson 和百度的小度机器人等智能产品的问世，人工智能成为大众热烈讨论的焦点。深度学习作为其中的核心技术之一，经过学术界与工业界的积极推动，已经被广泛应用于计算机视觉、语音识别和自然语言处理等诸多领域。如果读者想了解深度学习的总体概况，可参考作者编写的《深度学习导论及案例分析》^①一书。

本书专注讨论深度学习中应用非常广泛的模型——卷积神经网络，该模型特别适用于图像分类和识别、目标分割和检测以及人工智能游戏方面，受众对象包括计算机、自动化、信号处理、机电工程、应用数学等相关专业的研究生、教师以及算法工程师和科研工作者。

卷积神经网络是一种特殊的多层感知器或前馈神经网络，具有局部连接、权值共享的特点，其中大量神经元按照一定方式组织起来对视野中的交叠区域产生反应。其前身是日本学者 Fukushima 在感受野概念的基础上提出的神经认知机模型。利用神经认知机的思想，LeCun 等人在 1998 年提出了卷积神经网络的现代雏形 LeNet。2012 年，Krizhevsky 等人取得了卷积神经网络研究的重大突破，提出了著名的 AlexNet。AlexNet 在 ImageNet 的大规模图像分类竞赛中取得优异成绩，为深度学习的全面推广立下了汗马功劳。随后，卷积神经网络模型如雨后春笋般出现，如 VGGNet、GoogLeNet、SPPNet、ResNet、DenseNet、Faster R-CNN、YOLO、SSD、FCN、PSPNet、Mask R-CNN、SiameaseNet、SqueezeNet、DCGAN、NIN，以及在人工智能游戏中用到的深度强化模型等。

本书的最大特色是对卷积神经网络进行由浅入深的分类描述，依次包括：现代雏形、突破模型、应变模型、加深模型、跨连模型、区域模型、分割模型、特殊模型、强化模型和顶尖成就。这种分类框架是在模型概述和预备知识的基础上逐步展开的，既方

① 此书已由机械工业出版社出版，书号为 ISBN 978-7-111-55075-4。——编辑注



IV

便读者入门学习，又有助于读者深入钻研。

本书的另一大特色是结合 Caffe 或 TensorFlow 的代码来说明各种卷积神经网络模型的具体实现过程，并通过应用案例说明其价值和意义所在。典型的应用案例包括：字符识别、交通标志识别、交通路网提取、大规模图像分类、人脸图像性别分类、图像目标检测、图像语义分割、图像实例分割、人脸图像生成、Flappy Bird 智能体、AlphaGo 的仿效围棋程序等。读者可以通过运行各个应用案例的程序代码和实验数据，检验其演示效果。

与其他深度学习的书籍相比，本书对卷积神经网络的内容涵盖更为广泛、模型讨论更为深入、应用实践更为细致。特别是，还总结了一些运行卷积神经网络的配置技巧和操作经验。比如，在运行 Mask R-CNN 的时候，需要先安装读取 COCO 数据集的程序，然后再进行训练或测试。在运行 SSD 的时候，可视化结果只给出了类别编号而没有给出类别名，作者对此已进行了修改，以方便读者按照书中所示代码显示相应的类别名。把这些经过摸索得到的技巧和经验分享给读者，对提高读者的深度学习技术水平，无疑具有很好的加速作用。

本书是集体努力的成果，主要作者包括北京工业大学的李玉鑑、张婷、单传辉、刘兆英、聂小广和欧军。他们对全书的内容进行了精心的布局、认真的编写和细致的整理。同时，曾少锋、刘博文、穆红章、余华擎和方皓达等人在文献资料、实现代码和软件工具的收集方面也提供了积极的帮助。此外，华章公司的温莉芳副总经理和张梦玲编辑对本书的排版提出了许多宝贵的意见。最后，需要特别感谢家人的支持，他们也在不知不觉中以各种方式对此书出版做出了贡献。

限于作者水平，本书难免在内容取材和结构编排上有不妥之处，希望读者不吝赐教，提出宝贵的批评和建议，我们将不胜感激。

作 者

2018 年 4 月于北京工业大学



CONTENTS

目 录

前言

第 1 章 概述 1

1.1 深度学习的起源和发展 1

1.2 卷积神经网络的形成和演变 4

1.3 卷积神经网络的应用和影响 6

1.4 卷积神经网络的缺陷和视图 9

1.5 卷积神经网络的 GPU 实现和
cuDNN 库 10

1.6 卷积神经网络的平台和工具 10

1.7 本书的内容结构和案例数据 13

1.7.1 内容结构 13

1.7.2 案例数据 15

第 2 章 预备知识 22

2.1 激活函数 22

2.2 矩阵运算 23

2.3 导数公式 24

2.4 梯度下降算法 25

2.5 反向传播算法 26

2.5.1 通用反向传播算法 27

2.5.2 逐层反向传播算法 28

2.6 通用逼近定理 31

2.7 内外卷积运算 31

2.8 膨胀卷积运算 32

2.9 上下采样运算 33

2.10 卷积面计算 34

2.11 池化面计算 36

2.12 局部响应归一化 36

2.13 权值偏置初始化 37

2.14 丢失输出 37

2.15 丢失连接 38

2.16 随机梯度下降算法 39

2.17 块归一化 39

2.18 动态规划算法 40

第 3 章 卷积神经网络的现代
雏形——LeNet 41

3.1 LeNet 的原始模型 41

3.2 LeNet 的标准模型 43

3.3 LeNet 的学习算法 44

3.4 LeNet 的 Caffe 代码实现及说明 46

3.5 LeNet 的手写数字识别案例 54

3.6 LeNet 的交通标志识别案例 58

3.6.1 交通标志数据集的格式
转换 58

3.6.2 交通标志的识别分类 60



3.7	LeNet 的交通路网提取案例	63
3.7.1	交通路网的人工标注	64
3.7.2	交通路网的图像块分类	67
3.7.3	交通路网的图像块分类 LeNet	69
3.7.4	交通路网的自动提取 代码及说明	71
3.7.5	交通路网的自动提取程序 运行结果	75

第 4 章 卷积神经网络的突破 模型

4.1	AlexNet 的模型结构	78
4.2	AlexNet 的 Caffe 代码实现及 说明	82
4.3	AlexNet 的 Caffe 大规模图像分类 案例及演示效果	95
4.4	AlexNet 的 TensorFlow 代码 实现及说明	97
4.5	AlexNet 的 TensorFlow 大规模 图像分类案例及演示效果	103
4.6	AlexNet 的改进模型 ZFNet	107

第 5 章 卷积神经网络的应变 模型

5.1	SPPNet 的模型结构	109
5.2	SPPNet 的 Caffe 代码实现及 说明	112
5.3	SPPNet 的大规模图像分类 案例及演示效果	114

第 6 章 卷积神经网络的加深 模型

6.1	结构加深的卷积网络 VGGNet	118
-----	------------------	-----

6.1.1	VGGNet 的模型结构	118
6.1.2	VGGNet 的 TensorFlow 代码实现及说明	120
6.1.3	VGGNet 的物体图像分类 案例	129
6.2	结构更深的卷积网络 GoogLeNet	130
6.2.1	GoogLeNet 的模型结构	130
6.2.2	GoogLeNet 的 TensorFlow 代码实现及说明	136
6.2.3	GoogLeNet 的鲜花图像 分类案例	149

第 7 章 卷积神经网络的跨连 模型

7.1	快速网络 HighwayNet	154
7.2	残差网络 ResNet	155
7.2.1	ResNet 的模型结构	155
7.2.2	ResNet 的 Caffe 代码实现 及说明	157
7.2.3	ResNet 的大规模图像分类 案例	163
7.3	密连网络 DenseNet	169
7.3.1	DenseNet 的模型结构	169
7.3.2	DenseNet 的 Caffe 代码 实现及说明	171
7.3.3	DenseNet 的物体图像分类 案例	174
7.4	拼接网络 CatNet	178
7.4.1	CatNet 的模型结构	178
7.4.2	CatNet 的 Caffe 代码 实现及说明	179
7.4.3	CatNet 的人脸图像性别 分类案例	183



第 8 章 卷积神经网络的区域

模型 190

8.1 区域卷积网络 R-CNN 190

8.2 快速区域卷积网络 Fast R-CNN 191

8.3 更快区域卷积网络 Faster R-CNN 193

8.3.1 Faster R-CNN 的模型 结构 193

8.3.2 Faster R-CNN 的 Tensor Flow 代码实现及说明 196

8.3.3 Faster R-CNN 的图像目标 检测案例及演示效果 216

8.4 你只看一次网络 YOLO 220

8.4.1 YOLO 的模型结构 220

8.4.2 YOLO 的 TensorFlow 代码 实现及说明 226

8.4.3 YOLO 的图像目标检测 案例及演示效果 239

8.5 单次检测器 SSD 242

8.5.1 SSD 的模型结构 242

8.5.2 SSD 的 TensorFlow 代码 实现及说明 245

8.5.3 SSD 的图像目标检测 案例及演示效果 260

第 9 章 卷积神经网络的分割

模型 266

9.1 全卷积网络 FCN 266

9.1.1 FCN 的模型结构 266

9.1.2 FCN 的 Caffe 代码 实现及说明 269

9.1.3 FCN 的图像语义和几何

分割案例 272

9.2 金字塔场景分析网络 PSPNet 277

9.2.1 PSPNet 的模型结构 277

9.2.2 PSPNet 的 TensorFlow 代码实现及说明 282

9.2.3 PSPNet 的图像语义分割 案例及演示效果 291

9.3 掩膜区域卷积网络 Mask

R-CNN 294

9.3.1 Mask R-CNN 的模型 结构 294

9.3.2 Mask R-CNN 的 Keras 和 TensorFlow 代码实现及 说明 297

9.3.3 Mask R-CNN 的图像实例 分割案例及演示效果 318

第 10 章 卷积神经网络的特殊

模型 325

10.1 孪生网络 SiameseNet 325

10.1.1 SiameseNet 的模型 结构 325

10.1.2 SiameseNet 的 Caffe 代码实现及说明 326

10.1.3 SiameseNet 的手写数字 验证案例 328

10.2 挤压网络 SqueezeNet 331

10.2.1 SqueezeNet 的模型 结构 331

10.2.2 SqueezeNet 的 Caffe 代码实现及说明 334



10.2.3 SqueezeNet 大规模图像 分类案例	337
10.3 深层卷积生成对抗网络	
DCGAN	339
10.3.1 DCGAN 的模型结构	339
10.3.2 DCGAN 的 TensorFlow 代码实现及说明	340
10.3.3 DCGAN 的 CelebA 人脸 图像生成案例	345
10.4 网中网 NIN	348
10.4.1 NIN 的模型结构	348
10.4.2 NIN 的 Caffe 代码 实现及说明	350
10.4.3 NIN 大规模图像分类 案例	353

第 11 章 卷积神经网络的强化 模型

11.1 强化学习的基本概念	356
11.2 深度强化学习网络的学习算法	358
11.3 深度强化学习网络的变种模型	359
11.4 深度强化学习网络的 Flappy Bird 智能体案例	361
11.4.1 笨笨鸟网络的开发 环境和工具包	362
11.4.2 笨笨鸟网络的代码 实现及说明	363
11.4.3 笨笨鸟网络的学习训练 过程	367

11.4.4 笨笨鸟网络的演示 效果	370
-----------------------------	-----

第 12 章 卷积神经网络的顶尖 成就——AlphaGo

12.1 人工智能棋类程序简介	371
12.2 AlphaGo 的设计原理	373
12.2.1 总体思路	373
12.2.2 训练流程	374
12.2.3 搜索过程	377
12.3 AlphaGo Zero 的新思想	380
12.4 仿效 AlphaGo 的围棋程序 案例 MuGo	383
12.4.1 MuGo 的开发环境	383
12.4.2 MuGo 的代码实现及 说明	386
12.4.3 MuGo 的学习训练过程	401
12.4.4 MuGo 的演示效果	403

附录 A Caffe 在 Windows 上的 安装过程

附录 B Caffe 在 Linux 上的安装 过程

附录 C TensorFlow 在 Windows 上的安装过程

附录 D TensorFlow 在 Linux 上的安装过程

参考文献

概 述

深度学习是一种实现人工智能的强大技术，已经在图像视频处理、语音处理、自然语言处理等领域获得了大量成功的应用，并对学术界和工业界产生了非常广泛的影响。卷积神经网络是深度学习中最重要模型，2012 年以来极大地推进了图像分类、识别和理解技术的发展。而且通过与其他技术相结合，卷积神经网络还可用于设计实现游戏智能体 Q 网络、围棋程序 AlphaGo，以及语音识别和机器翻译软件等各种应用系统，所取得的成就已经使人工智能迈进了盛况空前、影响深远的新时代。本章主要介绍深度学习的起源和发展，说明卷积神经网络的形成和演变，分析卷积神经网络的应用和影响，讨论卷积神经网络的缺陷和视图，总结卷积神经网络的平台和工具，并概括本书的内容结构及案例数据。

1.1 深度学习的起源和发展

深度学习的概念起源于人工神经网络，本质上是指一类对具有深层结构的神经网络进行有效训练的方法。神经网络是一种由许多非线性计算单元（或称神经元、节点）组成的分层系统，通常网络的深度就是其中不包括输入层的层数。

最早的神经网络是心理学家 McCulloch 和数理逻辑学家 Pitts 在 1943 年建立的 MP 模型^[1]，如图 1.1 所示。MP 模型实际上只是单个神经元的形式化数学描述，具有执行逻辑运算的功能，虽然不能进行学习，但开创了人工神经网络研究的时代。1949 年，Hebb 首先对生物神经网络提出了有关学习的思想^[2]。1958 年，Rosenblatt 提出了感知器模型及其学习算法^[3]。在随后的几十年间，尽管神经网络的研究出现过一段与 Minsky 对感知器的批评有关的低潮期^[4]，但仍然在逐步向前推进，并产生了许多神经网络的新模型^[5-10]。到 20 世纪八九十年代，这些新模型终于引发了神经网络的重生，并掀起了对神经网络研究的世界性高潮^[11]。其中最受欢迎的模型至少包括：Hopfield 神经网络^[8]、波耳兹曼机^[9]和多层感知器^[10]。最早的深度学习系统也许就是那些通过数据分组处理方法训练的多层感知器^[12]。多层感知器，在隐含层数大于 1 时常称为深层感知器，实际上是一种由多层节点有向图构成的前馈神经网络^[13]，其中每一个非

输入节点是具有非线性激活函数的神经元，每一层与其下一层是全连接的。此外，Fukushima 提出的神经认知机可能是第一个具有“深度”属性的神经网络^[14-16]，并且也是第一个集成了“感受野”思想的神经网络^[17-18]，以便有效地对视觉输入的某些特性起反应。更重要的是，神经认知机促成了卷积神经网络结构的诞生和发展^[19]。而卷积神经网络作为一种判别模型，极大地推进了图像分类、识别和理解技术的发展，在大规模评测比赛中成绩卓著^[20]，盛誉非凡。

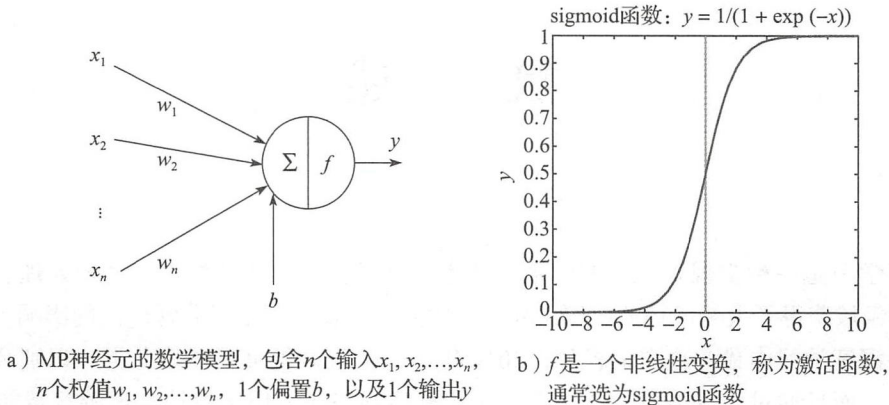


图 1.1

在训练神经网络方面，反向传播无疑是最常用、最著名的算法^[10, 21]。这是一种有监督学习算法，需要教师指导信号。也就是说，应提供一组训练样本，对给定的输入，指明相应的输出。然而，直到 20 世纪 80 年代末期，反向传播似乎还只是对浅层网络有效，尽管原理上也应对深层网络有效。浅层网络主要是指具有 1 个隐含层的神经网络，如图 1.2 所示。深层网络则主要是指具有 2 个及以上隐含层的神经网络，如图 1.3 所示。在早期的应用中，大多数多层感知器都只用 1 个或很少的隐含层，增加隐含层几乎没有什么经验上的收益。这似乎可以从神经网络的单隐层感知器逼近定理中找到某种解释^[22, 23]，该定理指出，只要单隐层感知器包含的隐含神经元足够多，就能够在闭区间上以任意精度逼近任何一个多变量连续函数。直到 1991 年的时候，关于多层感知器在增加层数时为什么难学习的问题，才开始作为一个深度学习的基本问题，得到了完全的理解。

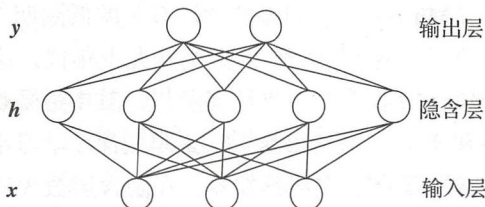


图 1.2 浅层（单隐层）神经网络

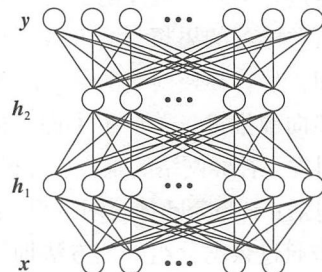


图 1.3 深层神经网络

1991年, Hochreiter正式指出, 典型的深层网络存在梯度消失或爆炸问题, 从而明确确立了深度学习的一个里程碑^[24]。该问题是说, 累积反向传播误差信号在神经网络的层数增加时会出现指数衰减或增长的现象, 从而导致数值计算快速收缩或越界。这就是深层网络很难用反向传播算法训练的主要原因。需要指出的是, 梯度消失或爆炸问题又称为长时滞后问题, 在循环神经网络中也会出现^[25]。

为了在一定程度上克服梯度消失或爆炸问题, 1990 ~ 2000年, Hochreiter的深邃思想推动了若干新方法的探索^[26-28]。但除了卷积神经网络以外^[29], 训练深层网络的问题直到2006年才开始得到严肃认真的对待。一个重要的原因是, 1995年之后支持向量机的快速发展减缓了神经网络的有关工作进展^[30]。

普遍认为, 深度学习正式发端于2006年, 以Hinton及其合作者发表的两篇重要论文为标志: 一篇发表在《Neural Computation》上, 题目为“A fast learning algorithm for deep belief nets”^[31]; 另一篇发表在《Science》上, 题目为“Reducing the dimensionality of data with neural networks”^[32]。从那以后, 大量的深度学习模型开始重新受到广泛关注, 或如雨后春笋般迅速发展起来, 其中主要包括受限波耳兹曼机(Restricted Boltzman Machine, RBM)^[33]、深层自编码器(deep AutoEncoder, deep AE)^[32]、深层信念网络(deep belief net)^[31]、深层波耳兹曼机(Deep Boltzman Machine, DBM)^[34]、和积网络(Sum-Product Network, SPN)^[35]、深层堆叠网络(Deep Stacked Network, DSN)^[36]、卷积神经网络(Convolutional Neural Network, CNN)^[19]、循环神经网络(Recurrent Neural Network, RNN)^[25]、长短期记忆网络(Long Short-Term Memory network, LSTM network)^[27]、强化学习网络(Reinforcement Learning Network, RLN)^[37]、生成对抗网络(Generative Adversarial Network, GAN)^[38]等。通过结合各种有效的训练技巧, 比如最大池化(max pooling)^[39]、丢失输出(dropout)^[40]和丢失连接(dropconnect)^[41], 这些深度学习模型取得了许多历史性的突破和成就, 例如手写数字识别^[32]、ImageNet分类^[20]和语音识别^[42]。而这些历史性的突破和成就, 使深度学习很快在学术界掀起了神经网络的一次新浪潮。其中最主要的原因, 当然是深度学习在解决大量实际问题时所表现的性能超越了机器学习的其他替代方法, 例如支持向量机^[30]。

在理论上, 一个具有浅层结构或层数不够深的神经网络虽然在节点数足够大时也可能充分逼近地表达任意的多元非线性函数, 但这种浅层表达在具体实现时往往由于需要太多的节点而无法实际应用。一般说来, 对于给定数目的训练样本, 如果缺乏其他先验知识, 人们更期望使用少量的计算单元来建立目标函数的“紧表达”, 以获得更好的泛化能力^[43]。而在网络深度不够时, 这种紧表达可能根本无法建立起来, 因为理论研究表明, 深度为 k 的网络能够紧表达的函数在用深度为 $k-1$ 的网络来表达时有时需要的计算单元会呈指数增长^[44]。这种函数表达的潜在能力说明, 深层神经网络(又称深度神经网络)在一定的条件下可能具有非常重要的应用前景。随着深度学习的兴起, 这种潜在能力开始逐步显现出来, 特别是对卷积神经网络的全面推广应用, 使得这种潜在能力几乎得到了淋漓尽致的发挥。

1.2 卷积神经网络的形成和演变

卷积神经网络最初是受到视觉系统的神经机制启发、针对二维形状的识别设计的一种生物物理模型，在平移情况下具有高度不变性，在缩放和倾斜情况下也具有一定的不变性。这种生物物理模型集成了“感受野”的思想，可以看作一种特殊的多层感知器或前馈神经网络，具有局部连接、权值共享的特点，其中大量神经元按照一定方式组织起来对视野中的交叠区域产生反应。1962年，Hubel和Wiesel通过对猫的视觉皮层细胞的研究，提出了感受野的概念^[17-18]。1979年，日本学者Fukushima在感受野概念的基础上，提出了神经认知机模型^[14-16]，该模型被认为是实现的第一个卷积神经网络。1989年，LeCun等人首次使用了权值共享技术^[45]。1998年，LeCun等人将卷积层和下采样层相结合，设计卷积神经网络的主要结构，形成了现代卷积神经网络的雏形（LeNet）^[19]。2012年，卷积神经网络的发展取得了历史性的突破，Krizhevsky等人采用修正线性单元（Rectified Linear Unit, ReLU）作为激活函数提出了著名的AlexNet，并在大规模图像评测中取得了优异成绩^[46]，成为深度学习发展史上的重要拐点。

在理论上，卷积神经网络是一种特殊的多层感知器或前馈神经网络。标准的卷积神经网络一般由输入层、交替的卷积层和池化层、全连接层和输出层构成，如图1.4所示。其中，卷积层也称为“检测层”，“池化层”又称为下采样层，它们可以被看作特殊的隐含层。卷积层的权值也称为卷积核。虽然卷积核一般是需要训练的，但有时也可以是固定的，比如直接采用Gabor滤波器^[47]。作为计算机视觉领域最成功的一种深度学习模型，卷积神经网络在深度学习兴起之后已经通过不断演化产生了大量变种模型。

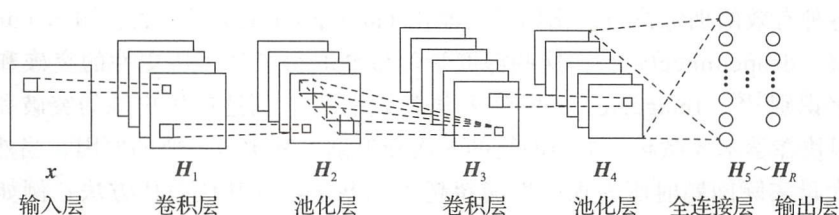


图 1.4 标准卷积神经网络

从结构的角度看，卷积神经网络起初只能处理黑白或灰度图像，变种模型通过把红、绿、蓝3个颜色通道作为一个整体输入已能直接处理彩色图像^[46]，有些还可以直接处理多帧图像甚至连续图像^[48]。同时，变种模型可以使用多个相邻的卷积层或多个相邻的池化层，也可以使用重叠池化和最大池化，还可以使用修正线性单元、渗漏修正线性单元（Leaky ReLU, LReLU）、参数修正线性单元（Parametric ReLU, PReLU）或指数线性单元（Exponential Linear Unit, ELU）取代sigmoid单元作为激活函数^[46, 49-51]，也可以在输出层采用软最大函数softmax替代sigmoid函数以产生伪概率。此外，卷积神经网络可以设计成孪生结构（siamese architecture），把原始数据映射到目标空间，产生对几何扭曲的鲁棒性^[52]。最后，

卷积神经网络可以设计成快道结构,允许信息通过快道无阻碍地跨越多层流动,使得用梯度下降训练非常深的网络变得更加容易^[53]。

从卷积核的角度看,卷积神经网络可以通过采用非常小的卷积核,比如 1×1 和 3×3 大小,被加深成一个更深的网络,比如16层或19层的VGGNet^[54]。如果采用参数修正线性单元代替修正线性单元,可以把VGGNet发展成MSRANet^[55]。而且,卷积神经网络通过使用小卷积核在保持总体计算代价的条件下增加深度和宽度,并与“摄入模块(inception module)”进行集成,可以用来建立谷歌网络(GoogLeNet)^[56]。此外,卷积神经网络通过使用微型多层感知器代替卷积核,还可以被扩展成更为复杂的网络,例如“网中网(Network In Network, NIN)”^[57]。

从区域的角度看,区域卷积神经网络(Region-based CNN, R-CNN)可以用来抽取区域卷积特征,并通过区域提议进行更加鲁棒的定位和分类^[58]。空间金字塔池化网络(Spatial Pyramid Pooling Net, SPPNet)可以克服其输入大小固定的缺点,办法是在最后一个卷积层和第一个全连接层之间插入一个空间金字塔池化层^[59]。不管输入的大小如何,空间金字塔池化层都能够产生固定大小的输出,并使用多尺度空间箱(spatial bin)代替滑动窗口对在不同尺度上抽取的特征进行池化。虽然与R-CNN相比,空间金字塔池化网络具有能够直接输入可变大小图像的优势,但是它们需要一个多阶段的管道把特征写入硬盘,训练过程较为麻烦。为了解决这个训练问题,可以在R-CNN中插入一个特殊的单级空间金字塔池化层(称为感兴趣区池化层,ROI pooling layer),并将其提取的特征向量输入到一个最终分化成两个兄弟输出层的全连接层,再构造一个单阶段多任务损失函数对所有网络层进行整体训练,建立快速区域卷积神经网络(Fast R-CNN)^[60],其优点是可以通过优化一个单阶段多任务损失函数进行联合训练。为了减少区域提议的选择代价,可以插入一个区域提议网络与Fast R-CNN共享所有卷积层,进一步建立更快速区域卷积神经网络(Faster R-CNN),产生几乎零代价的提议预测对象(或称为目标、物体)边界及有关分数^[61]。为了获得实时性能极快的对象检测速度,可以把输入图像划分成许多网格,并通过单个网络构造的整体检测管道,直接从整幅图像预测对象的边框和类概率建立YOLO模型,只需看一遍图像就能知道对象的位置和类别^[62]。为了更准确地定位对象,可以在多尺度特征图的每个位置上,使用不同长宽比的缺省框建立单次检测器(SSD)来取代YOLO^[63]。此外,采用空间变换模块有助于卷积神经网络学到对平移、缩放、旋转和其他扭曲更鲁棒的不变性^[64]。最后,可以把Faster R-CNN扩展成掩膜区域卷积神经网络(Mask R-CNN),在图像中有效检测对象的同时,还能够对每个对象实例生成一个高质量的分割掩膜^[65]。

从优化的角度看,许多技术可以用来训练卷积神经网络,比如丢失输出^[40, 66]、丢失连接^[41]和块归一化(batch normalization)^[67]。丢失输出是一种减小过拟合的正则化技术,而丢失连接是丢失输出的推广。块归一化(或批量归一化)则是按迷你块大小对某些层的输入进行归一化处理的方法。此外,残差网络(Residual Network, ResNet)采用跨越2~3层的连接策略也是一种重要的优化技术,可以用来克服极深网络的训练困难。借助残差学习能

够快速有效地成功训练超过 150 层甚至 1000 层的深层卷积神经网络，它在 ILSVRC & COCO 2015 的多项任务评测中发挥了关键作用^[68]，全部取得了第一名的突出成绩。最后，为了优化模型的结构，还可以采用火焰模块（fire module）建立卷积神经网络的挤压模型 SqueezeNet^[69]，也可以结合深度压缩（deep compression）技术进一步减少网络的参数^[70]。

从模型演变的角度看，卷积神经网络的发展脉络如图 1.5 所示。从图中可以看出，现代卷积网络以 LeNet 为雏形，在经过 AlexNet 的历史突破之后，演化生成了很多不同的网络模型，主要包括：加深模型、跨连模型、应变模型、区域模型、分割模型、特殊模型和强化模型等。加深模型的代表是 VGGNet-16、VGGNet-19 和 GoogLeNet；跨连模型的代表是 HighwayNet、ResNet 和 DenseNet；应变模型的代表是 SPPNet；区域模型的代表是 R-CNN、Fast R-CNN、Faster R-CNN、YOLO 和 SSD；分割模型的代表是 FCN、PSPNet 和 Mask R-CNN；特殊模型的代表是 SiameseNet、SqueezeNet、DCGAN、NIN；强化模型的代表是 DQN 和 AlphaGo。

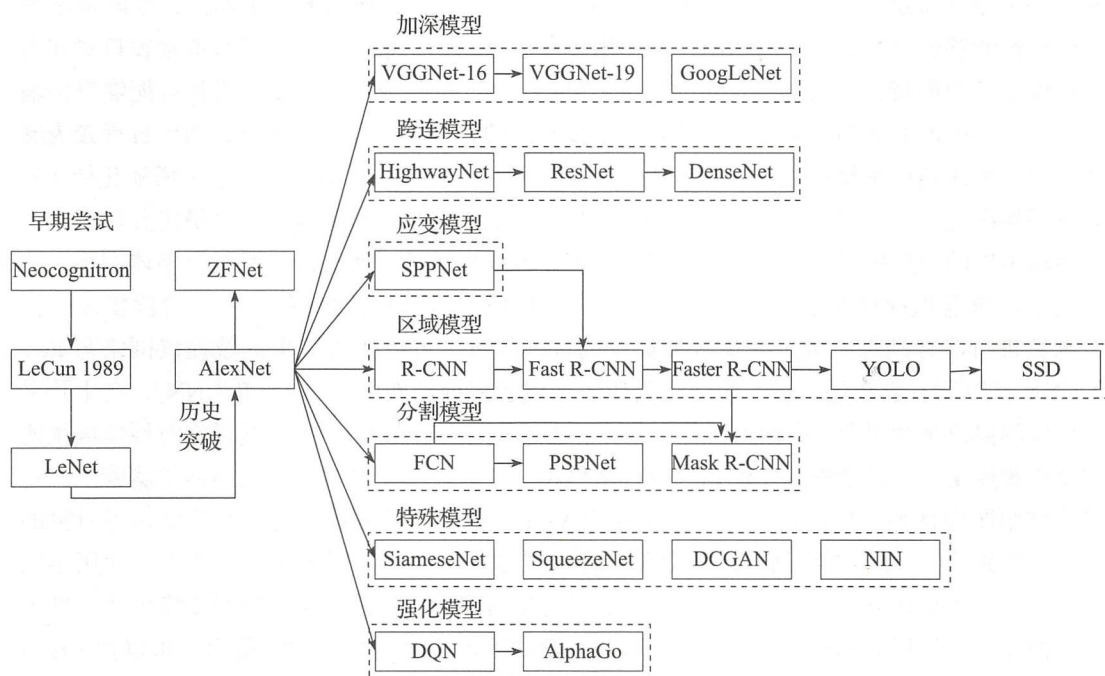


图 1.5 卷积神经网络的演变

1.3 卷积神经网络的应用和影响

自从卷积神经网络在深度学习领域闪亮登场之后，很快取得了突飞猛进的进展，不仅显著提高了手写字符识别的准确率，而且屡屡在图像分类与识别、目标定位与检测等大规模数据评测竞赛中名列前茅、战绩辉煌。此外，卷积神经网络在人脸验证、交通标志识别、视

频游戏、视频分类、语音识别、机器翻译、围棋程序等各个方面也获得广泛的成功应用。

在手写字符识别方面, LeCun 等人早在 1998 年就采用卷积神经网络模型使 MNIST 数据集上的错误率达到了 0.95% 以下^[19], Simard 等人在 2003 年采用交叉熵训练卷积神经网络把 MNIST 数据集上的错误率进一步降到了 0.4%, Ranzato 等人在 2006 年采用大卷积神经网络和无监督预训练又把 MNIST 数据集上的错误率降到了 0.39%, Ciresan 等人在 2012 年采用卷积神经网络的委员会模型把 MNIST 数据集上的错误率降到了目前的最低水平 0.23%。更详细的统计结果请访问网址 <http://yann.lecun.com/exdb/mnist/>。

在图像分类方面, 由 Krizhevsky、Sutshever 和 Hinton 组织的超级视觉队 (SuperVision) 于 2012 年实现了一个深层卷积神经网络, 参加大规模视觉识别挑战赛 (ImageNet Large Scale Visual Recognition Challenge 2012, ILSVRC-2012) 时获得了最好的前 5 测试错误率 (16.4%), 比第二名的成绩低 10% 左右^[46]。这个卷积神经网络现在称为 AlexNet, 使用了 “dropout” 优化技术和 “ReLU” 激活函数, 以及非常有效的 GPU 实现, 显著加快了训练过程。2013 ~ 2017 年的挑战赛中, 成绩最好的图像分类系统分别是 Claeifai^[71]、GoogLeNet^[56]、残差网络^[68]、六模型集成 (ensemble of 6 model)^[72]、双通道网络 (Dual Path Network, DPN)^[73], 它们都使用了卷积神经网络的模型结构。这些网络获得的前 5 测试错误率分别为 11.7%、6.7%、3.57%、2.99% 和 3.41%。

在 ILSVR 2012 ~ 2017 年的单目标定位挑战赛上, 获得最好错误率的系统都集成了卷积神经网络, 分别是 AlexNet^[46]、Overfeat^[74]、VGGNet^[54]、ResNet^[68]、集成模型 3 (ensemble 3)^[72] 和双通道网络^[73], 相应的最好错误率分别为 34.2%、29.9%、25.3%、9.02%、7.71% 和 6.22%。在 ILSVRC-2014 的目标检测挑战赛上, Lin 等人将 R-CNN 和 NIN 相结合, 获得了 37.2% 的平均准确率^[57], Szegedy 等人使用 GoogLeNet 获得了 43.9% 的平均准确率^[56]。在 ILSVRC-2015 的目标检测挑战赛上, He 等人将 Faster R-CNN 和 ResNet 相结合, 获得了 62.1% 的平均准确率, 比第二名高出了 8.5%^[68]。在 2016 年的目标检测挑战赛上, Zeng 等人采用门控双向卷积神经网络 (gated bi-directional CNN) 获得了 66.28% 的平均准确率^[75]。在 2017 年的目标检测挑战赛上, Shuai 等人将特征金字塔网络与门控双向卷积神经网络相结合, 获得了 73.14% 的平均准确率。

在人脸验证方面, Fan 等人于 2014 年建立了一个金字塔卷积神经网络 (pyramid CNN), 在 LFW 数据集上获得了 97.3% 的准确率, 其中 LFW 是 “Labeled Faces in the Wild” 的缩写^[76]。2015 年, Ding 等人利用精心设计的卷积神经网络和三层堆叠的自编码器建立了一个复杂的混合模型, 在 LFW 数据集上获得了高于 99.0% 的准确率^[77]。Sun 等人提出了一个由卷积层和摄入层 (inception layer) 堆叠而成的 DeepID3 模型, 在 LFW 数据集上获得了 99.53% 的准确率^[78]。此外, Schroff 等人实现了 “FaceNet” 系统, 在 LFW 和 YouTube 人脸数据集上分别获得了 99.63% 和 95.12% 的准确率^[79]。

在交通标志识别方面, Ciresan 等人于 2011 年实现了一个由卷积神经网络和多层感知器构成的委员会机器, 在德国交通标志识别标准数据集 (German Traffic Sign

Recognition Benchmark, GTSRB) 上获得了 99.15% 的准确率^[80]。2012 年, Ciresan 等人提出了一个多列卷积神经网络, 在 GTSRB 上获得了 99.46% 的准确率, 超过了人类的识别结果^[81]。

在视频游戏方面, Mnih 等人于 2015 年通过结合卷积神经网络和强化学习, 开发了一个深度 Q- 网络智能体的机器玩家^[37], 只需输入场景像素和游戏得分进行训练, 就能够让很多经典的 Atari 2600 视频游戏成功学会有效的操作策略, 达到与人类专业玩家相当的水平。这种深度 Q- 网络智能体在高维感知输入和行为操纵之间的鸿沟上架起了一座桥梁, 能够出色地处理各种具有挑战性的任务。

在视频分类方面, 使用独立子空间分析 (Independent Subspace Analysis, ISA) 方法, Le 等人于 2011 年提出了堆叠卷积 ISA 网络, 能够从无标签视频数据中学习不变的时空特征。该网络在 Hollywood 2 和 YouTube 数据集上分别获得了 53.3% 和 75.8% 的准确率^[82]。2014 年, Karpathy 等人对卷积神经网络在大规模视频分类上的效果进行了广泛的经验评估, 在 Sports-1M 测试集的 200 000 个视频上获得了 63.9% 的 Hit@1 值 (即前 1 准确率)^[83]。2015 年, Ng 等人采用卷积神经网络和长短期记忆循环神经网络的混合模型, 在 Sports-1M 测试集上获得了 73.1% 的 Hit@1 值^[84]。

在语音识别方面, Abdel-Hamid 等人于 2012 年第一次证实, 使用卷积神经网络能够在频率坐标轴上有效归一化说话人的差异, 并在 TIMIT 音素识别任务上将音素错误率从 20.7% 降到 20.0%^[85]。这些结果在 2013 年被微软研究院的 Abdel-Hamid 等人 and Deng 等人以及 IBM 研究院的 Sainath 等人使用改进的卷积神经网络结构、预训练和池化技术拓展到大词汇语音识别上^[86-87]。进一步的研究表明, 卷积神经网络对训练集或者数据差异较小的任务帮助最大^[88-90]。此外, 通过结合卷积神经网络、深度神经网络和基于 i-vector 的自适应技术, IBM 的研究人员在 2014 年说明他们能够将 Switchboard Hub5'00 评估集的词错误率降至 10.4%。

在机器翻译方面, Gehring 等人使用一种全新的卷积神经网络模型进行从序列到序列的学习^[91], 能够在非常大的标准数据集上超越循环神经网络的性能, 不仅可以大幅提高翻译速度, 同时也提高了翻译质量。比如, 这种全新的模型在 WMT'16 英语到罗马尼亚语的翻译任务上可比以前最好的系统提高 1.8 的 BLEU 分数, 在 WMT'14 英语到法语的翻译任务上可比 Wu 等人的长短期记忆神经翻译模型提高 1.5 的 BLEU 分数^[92], 在 WMT'14 英语到德语的翻译任务上可超过当前最高水平 0.5 的 BLEU 分数。

在围棋程序方面, DeepMind 开发的 AlphaGo 利用深层网络和蒙特卡罗树搜索 (Monte Carlo tree search), 2015 年 10 月首次在完整的围棋比赛中没有任何让子以 5 比 0 战胜了人类的专业选手、欧洲冠军、职业围棋二段选手樊麾^[93], 这也是计算机围棋程序首次击败围棋职业棋手。2016 年 3 月, AlphaGo 又以 4 比 1 战胜了人类的顶尖高手、世界冠军、职业围棋九段选手李世石。2016 年末 2017 年初, AlphaGo 在中国棋类网站上以 Master 为注册账号与中日韩数十位围棋高手进行快棋对决, 连续 60 局无一败绩。

2017年5月,在中国乌镇围棋峰会上,AlphaGo以3比0战胜排名世界第一的围棋冠军柯洁。

1.4 卷积神经网络的缺陷和视图

从上述应用和成果不难看出,卷积神经网络已经使人工智能迈进了盛况空前、影响深远的新时代。不过这并不等于说,可以用卷积神经网络完全实现人类的智能。虽然现在卷积神经网络分类图像中的对象能够达到与人类匹敌的水平^[68],但其视觉与人类的视觉相比仍然是非常不同的^[94]。事实上,即使成功训练之后,卷积神经网络也仍然可能错分对抗样本。对抗样本是一种含有人类不可感知的微小扰动的非随机图像,如图1.6所示,在一幅熊猫图像中加入微量噪声后,它可能变成一幅对抗熊猫图像的样本,人类仍然能够轻松识别它为熊猫,但卷积神经网络却一口咬定它是长臂猿,详情请参见文献[95]。另外,有些人类根本不能识别的噪声图像,如图1.7所示,却可能成为卷积神经网络的欺骗图像,让卷积神经网络以高于99%的置信度识别它为一个熟知的对象(比如数字)^[96]。因此,卷积神经网络在实际应用中仍然存在一些不易被察觉的潜在缺陷。



图 1.6 对抗图像样本举例

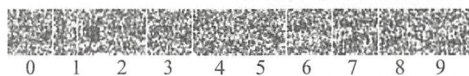


图 1.7 欺骗图像举例。随机噪声图像欺骗卷积神经网络,被识别为数字0~9

为了更好地理解卷积神经网络的成功与失败,一种办法是采用可视化技术来分析其数据表达并解释其工作机理^[97-98],例如以某种可见视图方式来显示激活和特征。通过可视化技术,能够按照逐级上升的顺序展现卷积神经网络各层的直觉期望性质,包括组合性、不变性和类别性。常用可视化技术,比如逆变换(inversion)、激活最大化(activation maximization)和卡通化(caricaturization),是以自然原像概念为基础的。自然原像就是那些看起来具有显著自然特征的图像。通常,一幅图像经过卷积神经网络提取特征后,随着层次的深入,可视化结果会变得越来越模糊和抽象^[98]。比如,图1.8是用AlexNet处理一幅狗的图像后可视化各层特征的结果,看起来逐层模糊和抽象。

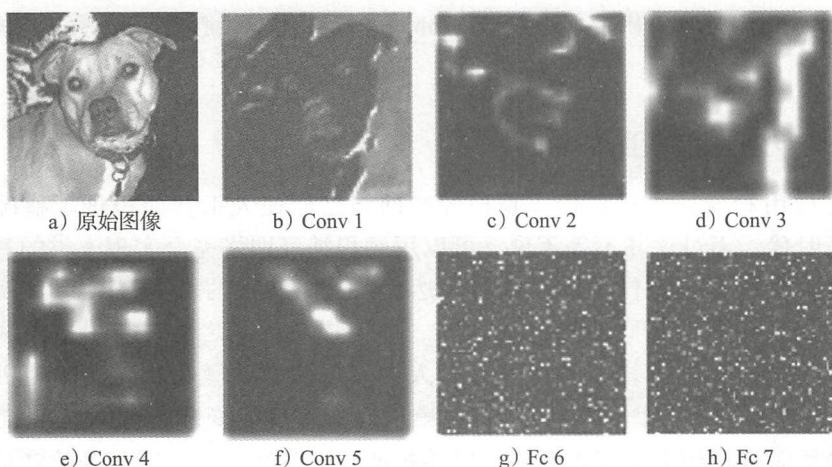


图 1.8 卷积神经网络的逐层可视化举例

1.5 卷积神经网络的 GPU 实现和 cuDNN 库

只采用 CPU 在大规模数据集中训练卷积神经网络的速度很慢，因此可以结合图形处理单元（Graphic Processing Unit, GPU）进行加速。GPU 具有单指令多数数据流结构，非常适合用一个程序处理各种大规模并行数据的计算问题。最常用的 GPU 是英伟达（Nvidia）生产的。编写 GPU 代码可在 CUDA 环境下进行。CUDA（Compute Unified Device Architecture）是一种用于 GPU 通用计算的并行计算平台和编程模型。它以 C 语言为基础，并对 C 语言进行了扩展，能够在显卡芯片上执行程序。CUDA 提供了一个深度神经网络的 GPU 加速库 cuDNN（CUDA Deep Neural Network），完成了对卷积、池化、归一化和激活函数层等标准操作的快速实现。如果读者想了解更多的相关信息，可参考以下网站：

- <https://developer.nvidia.com/cuda-toolkit>
- <https://developer.nvidia.com/deep-learning-software>
- <https://developer.nvidia.com/cudnn>

1.6 卷积神经网络的平台和工具

为了解决各种实际问题中有效地利用深度学习模型，特别是卷积神经网络，现在已经有很多开发平台和工具可以选择。比较常用的有 Theano、TensorFlow、Caffe、Caffe 2、CNTK、MXNet、Torch、Deeplearning4J 和 Keras 等，其中 TensorFlow、Caffe 2 和 MXNet 之间的竞争可能会比较激烈。目前，这些平台和工具还没有任何一种完善到足以解决“所有”的业务问题，大多通过专有解决方案提供先进的机器学习和人工智能的功能，包括手写字识别、图像识别、视频识别、语音识别、自然语言处理和对象识别等高级功能。下面分别对它们进行简要的说明。

1) Theano 由蒙特利尔大学学习算法学院的 30 ~ 40 名教师和学生集体维护, 其创始人是深度学习研究的重要贡献者 Yoshua Bengio。Theano 通过 BSD 许可发布, 支持快速开发高效的机器学习算法。Theano 的结构相当简单, 以 Python 为代码库和接口, 其中 C/CUDA 代码也被打包成 Python 字符串。这对开发者来说很难驾驭、调试和重构。Theano 开创了使用符号图来编程网络的趋势, 其符号 API 支持循环控制, 使得循环神经网络的实现更容易、更高效。虽然 Theano 是一个很好的学术研究工具, 在单个 CPU 上的运行效率较高, 但缺乏分布式应用程序管理框架, 只支持一种编程开发语言, 在开发大型分布式应用程序时可能会遇到挑战。

2) TensorFlow 来自早期的 Google 库 DistBelief V2, 是作为 Google Brain 项目的一部分开发的专有深度网络库。由于 TensorFlow 支持广泛的功能, 如图像识别、手写字符识别、语音识别、预测以及自然语言处理, 所以在 2015 年 11 月 9 日以 Apache 2.0 许可开源后, 谷歌立即获得了大量的关注。有些人评价 TensorFlow 是对 Theano 的重新设计。TensorFlow 在 2017 年 2 月 15 日发布了 1.0 版本, 是 8 个先前版本的累积, 解决了很多不完整的核心功能和性能问题。TensorFlow 的编程接口包括 Python 和 C++, 并支持 Java、Go、R 和 Haskell API 的 alpha 版本接口。另外, TensorFlow 支持精细的网格层, 允许用户构建新的复杂层类型, 允许模型的不同部分在不同的设备上并行训练, 还可以使用 C++ Eigen 库在 ARM 架构上编译和优化。经过训练的 TensorFlow 模型可以部署在各种服务器或移动设备上, 无须实现单独的解码器或加载 Python 解释器。

3) Caffe 开创于 2013 年年底, 可能是第一个主流的行业级深度学习工具包, 由领导 Facebook AI 平台工程的贾扬清负责设计和实现, 在 BSD 2-Clause 开源许可后发布。作为一种在计算机视觉界最受欢迎的工具包, Caffe 具有优良的卷积神经网络模型结构, 在 2014 年 ImageNet 挑战赛中脱颖而出。Caffe 的运行速度快, 学习速度为 4ms/图, 推理速度为 1ms/图, 在单个 Nvidia K40 GPU 上每天处理超过 6000 万张图片, 是研究实验和商业部署的完美选择。Caffe 是基于 C++ 的, 可以在各种跨平台设备上编译, 包括一个 Windows 的端口, 支持 C++、Matlab 和 Python 等编程接口。而且, Caffe 拥有一个庞大的用户社区为其深层网络存储库做贡献, 包括 AlexNet 和 GoogLeNet 两种流行的用户网络。Caffe 的缺点是不支持细粒度网络层, 在构建复合层类型时必须以低级语言完成, 对常规网络和语言建模的支持总体上很差。

4) Caffe 2 是 Caffe 的升级版, 于 2017 年 4 月 18 日由 Facebook 根据 BSD 许可协议开源, 继续强力支持视觉类型问题, 并增加了自然语言处理、手写识别和时间序列预测的循环神经网络和长短期记忆网络。Caffe 2 可以把 Caffe 模型轻松转换为实用程序脚本, 但更侧重于模块化、卓越的移动和大规模部署, 能够像 TensorFlow 一样使用 C++ Eigen 库来支持 ARM 架构, 并在移动设备上部署深度学习模型。随着 Facebook 最新宣布其改变航向, Caffe 2 在深度学习社区中为大众所热捧, 可能超越 Caffe 成为主要的深度学习框架。

5) CNTK 开始称为 Computational Network Toolkit (计算网络工具包), 但在 CNTK 2.0 Beta1 版本根据 MIT 许可发布后被正式更名为 Microsoft Cognitive Toolkit (微软认知套件)。

CNTK 最早是由微软的计算机科学家开发的，目的是想要更快、更高效地提高语音识别技术，但很快就超越了语音领域并演变成了一个产品，包括一些领先的国际家电制造商和微软的旗舰产品组在内的客户依靠它来执行各种各样的深度学习任务。CNTK 可以运行在使用传统 CPU 或 GPU 的计算机上，既可以运行在一台笔记本电脑上，也可以运行在数据中心的一系列计算机上，支持使用 Python 或 C++ 编程接口的 64 位 Linux 和 Windows 操作系统，能够轻松处理从相对较小到非常巨大等各种规模的数据集。与 TensorFlow 和 Theano 的组成相似，CNTK 的网络被描述为向量运算（如矩阵的加法 / 乘法或卷积）的符号图，允许用户构建细粒度的网络层并创造新的复合层类型，而不像 Caffe 那样需要通过低级语言实现。此外，CNTK 又有点类似 Caffe，也是基于 C++ 的，具有跨平台的 CPU/GPU 支持，并在 Azure GPU Lab 提供了最高效的分布式计算性能。目前，CNTK 的主要不足是对 ARM 架构缺乏支持，这限制了其在移动设备上的功能。

6) MXNet (发音为 “mix-net”) 起源于卡内基 - 梅隆大学和华盛顿大学，2017 年 1 月 30 日进入 Apache 基金会成为孵化器项目，是一个功能齐全、可编程和可扩展的深度学习框架，支持各种深度学习模型（比如卷积神经网络、循环神经网络和长短期记忆网络），也是目前唯一支持生成对抗网络模型的深度学习框架。而且，MXNet 提供了混合编程模型（命令式和声明式）的功能、大量编程语言的代码（包括 Python、C++、R、Scala、Julia、Matlab 和 JavaScript），以及强大的扩展能力（如 GPU 并行性和内存镜像、编程器开发速度和可移植性），甚至被有些人称为世界上最好的图像分类器。此外，MXNet 与 Apache Hadoop YARN（一种通用的、分布式的应用程序管理框架）集成，使其成为 TensorFlow 的竞争对手。特别是，亚马逊首席技术官 Werner Vogels 选择公开支持 MXNet，苹果公司在 2016 年收购 Graphlab/Dato /Turi 之后也传闻使用它。

7) Torch 的主要贡献者是 Facebook、Twitter 和 Nvidia，Google Deep Mind 也有一部分功劳。Torch 按 BSD 3 开源许可发布，以非主流编程语言 Lua 实现，在员工熟练掌握之前很难提高整体效率，限制了其广泛应用。当前的版本 Torch7 提供了一个比 Caffe 更详尽的接口库，可以在上面非常方便地对已有模块实现逻辑复杂的调用。相比于 Caffe，Torch7 开放的接口更多，使用更灵活，很少会通过开发者给它实现新功能，而是依赖它去做扩展。但由于 Lua 语言本身功能偏弱，有点先天不足，所以 Torch7 不适合做层本身的组件开发。此外，Torch 缺乏 TensorFlow 的分布式应用程序管理框架。

8) Deeplearning4J，简称 DL4J，是用 Java 和 Scala 编写的、由 Apache 2.0 授权的开放源码，支持常用的机器学习向量化工具，以及丰富的深度学习模型，包括受限波耳兹曼机、深信度神经网络、卷积神经网络、循环神经网络、RNTN 和长短期记忆网络等。DL4J 是 SkyMind 的 Adam Gibson 的创意，是唯一与 Hadoop 和 Spark 集成的商业级深度学习框架，内置多 GPU 支持，可协调多个主机线程，使用 Map-Reduce 来训练网络，同时依靠其他库来执行大型矩阵操作。DL4J 在 Java 中开源，本质上比 Python 快，速度与 Caffe 相当，可以实现多个 GPU 的图像识别、欺诈检测和自然语言处理等出色功能。

9) Keras 是一个高层神经网络的应用程序编程接口 (Application Programming Interface, API), 由纯 Python 语言编写, 并且使用 TensorFlow、Theano 或者 CNTK 作为后端。Keras 的设计有 4 个原则: 用户友好、模块性、易扩展性和与 Python 协作。用户友好是指 Keras 提供一致而简洁的 API, 以及清晰而有用的 bug 反馈, 极大地减少了用户工作量。模块性是指 Keras 将网络层、损失函数、优化器、激活函数等方法都表示为独立的模块, 作为构建各种模型的基础。易扩展性是指在 Keras 中只需要仿照现有的模块编写新的类或函数即可添加新的模块, 非常方便。与 Python 协作是指 Keras 没有单独的模型配置文件, 模型完全由 Python 代码描述, 具有更紧凑和更易调试的优点。

如果读者想了解上述开发工具的更多信息和资料, 可以访问下面的网址:

- <http://www.deeplearning.net/software/theano/>
- <http://tensorflow.org>
- <http://caffe.berkeleyvision.org>
- <https://developer.nvidia.com/caffe2>
- <http://cntk.codeplex.com/>
- <http://mxnet.io>
- <http://torch.ch/>
- <http://deeplearning4j.org>
- <https://keras.io/>

此外, 必须介绍一下 CUDA-convnet。这是一个 C++/CUDA 实现的高性能卷积神经网络库, 其中甚至包括更一般的前馈神经网络。目前有 CUDA-convnet 和 CUDA-convnet2 两个版本。CUDA-convnet 可以建立任意层的连通性和网络深度, 实现任何有向无环图, 使用反向传播算法进行训练, 需要 Fermi-generation GPU (GTX 4xx、GTX 5xx 或者 Teslax 系列)。在 CUDA-convnet 的基础上, CUDA-convnet2 主要做了 3 个改进。一是在 Kepler-generation Nvidia GPU 上提高了训练速度 (Geforce Titan、K20、K40); 二是实现了数据并行、模型并行和二者混合并行的方式^[99]; 三是改进了不太友好的代码, 完善了一些不完整的文档, 而且仍在不断补充完善。CUDA-convnet 和 CUDA-convnet2 的下载网址如下:

- <https://code.google.com/archive/p/cuda-convnet/downloads>
- <https://github.com/akrizhevsky/cuda-convnet2>

1.7 本书的内容结构和案例数据

卷积神经网络是目前应用最广的深度学习模型。本书旨在介绍其中比较重要的模型, 并通过演示案例说明有关模型的应用价值。下面简述本书的内容结构及案例数据。

1.7.1 内容结构

本书共分为 12 章, 有关应用案例的章节、框架和平台汇总在表 1.1 中。

表 1.1 应用案例的章节、框架和平台

章节	案例名称	使用框架	运行平台
3.5	LeNet 的手写数字识别案例	Caffe	Windows
3.6	LeNet 的交通标志识别案例	Caffe	Windows
3.7	LeNet 的交通路网提取案例	Caffe	Windows
4.3	AlexNet 的大规模图像分类案例	Caffe	Windows
4.5	AlexNet 的大规模图像分类案例	TensorFlow	Windows
5.3	SPPNet 的大规模图像分类案例	Caffe	Windows
6.1.3	VGGNet 的物体图像分类案例	TensorFlow	Windows
6.2.3	GoogLeNet 的鲜花图像分类案例	TensorFlow	Windows
7.2.3	ResNet 的大规模图像分类案例	Caffe	Windows
7.3.3	DenseNet 的物体图像分类案例	Caffe	Windows
7.4.3	CatNet 的人脸图像性别分类案例	Caffe	Windows
8.3.3	Faster R-CNN 的图像目标检测案例	TensorFlow	Linux
8.4.3	YOLO 的图像目标检测案例	TensorFlow	Windows
8.5.3	SSD 的图像目标检测案例	TensorFlow	Linux
9.1.3	FCN 的图像语义和几何分割案例	Caffe	Windows
9.2.3	PSPNet 的图像语义分割案例	TensorFlow	Linux
9.3.3	Mask R-CNN 的图像实例分割案例	Keras、TensorFlow	Windows
10.1.3	SiameseNet 的手写数字验证案例	Caffe	Windows
10.2.3	SqueezeNet 的大规模图像分类案例	Caffe	Windows
10.3.3	DCGAN 的人脸图像生成案例	TensorFlow	Windows
10.4.3	NIN 的大规模图像分类案例	Caffe	Windows
11.5	深度强化网络的 Flappy Bird 智能体案例	TensorFlow	Windows
12.4	仿效 AlphaGo 的围棋程序案例 MuGo	TensorFlow	Linux

各章的内容结构描述如下：

第 1 章为概述，介绍深度学习的起源和发展，说明卷积神经网络的形成和演变，分析卷积神经网络的应用和影响，讨论卷积神经网络的缺陷和视图，总结卷积神经网络的平台和工具。

第 2 章为预备知识，主要介绍卷积神经网络模型有关的数学基础。

第 3 章为卷积神经网络的现代雏形 LeNet。首先介绍 LeNet 的原始模型，然后描述 LeNet 的标准模型，接着给出 LeNet 的学习算法，说明 LeNet 的 Caffe 代码，并分析 LeNet 的手写数字识别案例、交通标志识别案例和交通路网提取案例。

第 4 章为卷积神经网络的突破模型 AlexNet。首先介绍 AlexNet 的模型结构，然后依次说明 AlexNet 的 Caffe 和 TensorFlow 代码，并分析 AlexNet 的大规模图像分类案例，最后简介其改进模型 ZFNet。

第 5 章为卷积神经网络的应变模型。主要介绍 SPPNet 的模型结构，说明 SPPNet 的 Caffe 代码，并分析 SPPNet 的大规模图像分类案例。

第 6 章为卷积神经网络的加深模型。主要介绍 VGGNet 和 GoogLeNet 的模型结构，说明它们的 TensorFlow 代码，并分析 VGGNet 的物体图像分类案例和 GoogLeNet 的鲜花图像

分类案例。

第7章为卷积神经网络的跨连模型。主要介绍快道网络、残差网络、密连网络和拼接网络。对于快道网络，只描述了模型结构。对于其余3个网络，还说明了核心模块的Caffe代码实现。此外，还分析了残差网络的大规模图像分类案例、密连网络的物体图像分类案例，以及拼接网络的人脸图像性别分类案例。

第8章为卷积神经网络的区域模型。主要介绍区域卷积神经网络、快速区域卷积网络、更快速区域卷积网络、你只看一次网络和单次检测器。对于前两个网络，只描述了模型结构。对于另外3个网络，还说明了它们的TensorFlow代码，并分析了它们在VOC 2007数据集上的图像目标检测案例。

第9章为卷积神经网络的分割模型。主要介绍全卷积网络、金字塔场景分析网络和掩膜区域卷积网络的模型结构，说明它们的Caffe或TensorFlow代码，并分析它们的图像语义分割、图像几何分割或图像实例分割等应用案例。

第10章为卷积神经网络的特殊模型。主要介绍4种模型，包括孪生网络、挤压网络、生成对抗网络和网中网。不仅给出了它们的模型结构，说明了它们的Caffe或TensorFlow代码，也分析了它们的手写数字验证、大规模图像分类或人脸图像生成等应用案例。

第11章为卷积神经网络的强化模型。主要介绍深层强化学习的标准模型、学习算法和变种模型，并分析一个笨笨鸟Flappy Bird智能体的游戏应用案例。

第12章为卷积神经网络的顶尖成就AlphaGo。主要介绍AlphaGo的设计原理和AlphaGo Zero的新思想，并分析一个仿效围棋程序MuGo的游戏应用案例。

1.7.2 案例数据

各章在介绍卷积神经网络的变种模型时，一般还给出了有关的应用案例。这些案例可能重复用到13个不同的数据集（大小信息和下载网址详见表1.2），分别是：MNIST（Mixed National Institute of Standard and Technology）、GTSRB（German Traffic Sign Recognition Benchmark）、RRSI（Road Remote Sensing Image）、ImageNet 2012、CIFAR-10、Oxford-17、AR、VOC 2007、SIFT Flow、ADE20K、COCO（Common Objects in Context）2014、CelebA和Gamerecords。

表 1.2 案例数据的信息描述和下载网址

序号	数据集名称	训练集	测试集	下载网址
1	MNIST	60 000	10 000	http://yann.lecun.com/exdb/mnist/
2	GTSRB	39 200	12 600	http://benchmark.ini.rub.de/?section=mydata&subsection=dataset
3	RRSI	11	5	http://www.escience.cn/people/guangliangcheng/Datasets.html
4	ImageNet 2012	1 281 167	50 000	http://www.cnblogs.com/zjutzz/p/6083201.html
5	CIFAR-10	50 000	10 000	http://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz
6	Oxford-17	1 088	272	http://www.robots.ox.ac.uk/~vgg/data/flowers/17/

(续)

序号	数据集名称	训练集	测试集	下载网址
7	AR	2 080	520	http://www2.ece.ohio-state.edu/~aleix/ARdatabase.html
8	VOC 2007	5 011	4 952	http://host.robots.ox.ac.uk/pascal/VOC/voc2007/index.html
9	SIFT Flow	2 488	200	http://www.cs.unc.edu/~jtighe/Papers/ECCV10/siftflow/SiftFlowDataset.zip
10	ADE20K	20 210	2 000	http://sceneparsing.csail.mit.edu/
11	COCO 2014	82 783	40 504	http://cocodataset.org/#download
12	CelebA	162 770	18 962	http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html
13	Gamerecords	4 491 800	100 000	https://u-go.net/gamerecords/

下面依次对每个数据集进行详细介绍。

1) MNIST 是一个著名的手写数字数据集 (如图 1.9 所示), 包括 60 000 个训练样本, 10 000 个测试样本。其中, 每个样本图像的大小为 28×28 像素, 仅包含一个单一的手写数字字符。像素的取值范围是 $[0, 255]$, 其中 0 表示黑, 255 表示白, 中间值表示灰度级。本书在第 3 章和第 10 章使用了 MNIST 数据集。

2) GTSRB 是一个德国交通标志数据集 (如图 1.10 所示)。其中有两套训练集和测试集, 都包含 43 类交通标志。一套有 39 209 个训练样本和 12 630 个测试样本, 另一套有 26 640 个训练样本和 12 569 个测试样本。本书在第 3 章的交通标志识别案例中选用了前一套训练集和测试集, 但从中去掉了少量样本, 只

用了 39 200 个训练样本和 12 600 个测试样本。GTSRB 的图片格式是 .ppm 类型, 大小在 15×15 到 250×250 之间不等, 每个样本的长宽、兴趣区和标签等注释信息存放在相应的 .csv 文件中。为了便于处理, 需要把它们的格式先转换成 .jpg 图像类型, 并归一化为 32×32 像素大小。

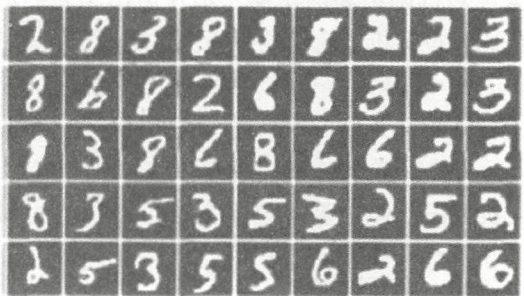


图 1.9 MNIST 的手写数字图像举例

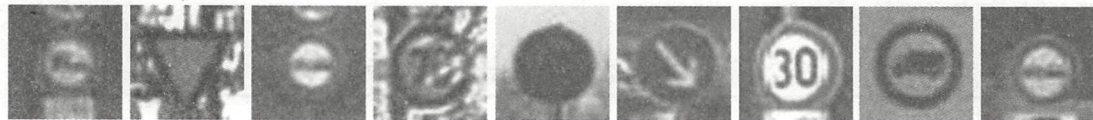


图 1.10 GTSRB 的交通标志图像举例

3) RRSI 是一个交通路网遥感图像的数据集 (如图 1.11 所示)。交通路网是指公路、城市道路和单位管辖范围允许社会机动车通行的地方, 包括广场、公共停车场等用于公众通行的场所。RRSI 实际上共有 30 幅大小不等的图像和 2 种标注。本书在第 3 章选用了 11 幅来训练, 5 幅来测试, 进行路网自动提取。



图 1.11 RRSI 的交通路网遥感图像举例

4) ImageNet 是一个拥有超过 1500 万幅图像、约 22 000 个类别的数据集 (如图 1.12 所示), 可用于大规模图像识别、定位和检测的研究。本书在第 4 章、第 5 章、第 7 章和第 10 章使用了 2012 年大规模图像视觉识别比赛 (Large Scale Visual Recognition Competition, ILSVRC) 的数据集 ImageNet (即 ImageNet 2012) 设计图像分类案例。ImageNet 2012 包含 1 281 167 幅训练图像和 50 000 幅测试图像, 共有 1000 个类别。训练集中各类图像的数目可能不同, 最少为 732 幅, 最多为 1300 幅。而测试集中各类图像的数目都是 50 幅。

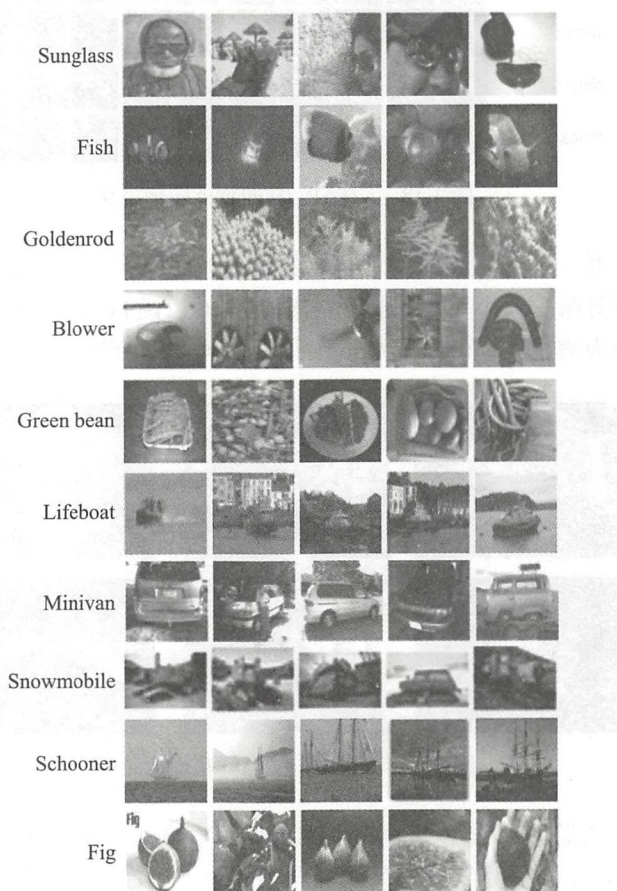


图 1.12 ImageNet 2012 的物体图像举例

5) CIFAR-10 是一个常见物体图像数据集 (如图 1.13 所示), 分为 10 个类别, 包含 60 000 幅 32×32 的彩色图像, 50 000 幅用于训练, 10 000 幅用于测试。注意: 根据表 1.2 中的网址下载的 CIFAR-10 是经过封装的, 有 3 个版本: Python、Matlab 和 Binary。本书在第 6 章和第 7 章选用了 Binary 版本。



图 1.13 CIFAR-10 的物体图像举例

6) Oxford-17 是一个鲜花图像数据集 (如图 1.14 所示), 其中包含 1360 幅图像, 分为 17 类, 每类 80 幅图像, 大小不尽相同。本书在第 6 章随机选了 1088 幅图像作为训练集, 其余 272 幅图像作为测试集, 设计了 GoogLeNet 的鲜花图像分类案例。



图 1.14 Oxford-17 的鲜花图像举例

7) AR 是一个人脸图像数据集 (如图 1.15 所示), 包含 126 个人在不同表情、光照和妆容条件下的 4000 多幅人脸图像, 但只能下载 100 个人的 2600 幅。本书在第 7 章的 CatNet 性别分类案例中从中选择了 40 名男性和 40 名女性的 2080 幅图像作为训练集, 其余的 520

幅图像作为测试集。



图 1.15 AR 的人脸图像举例

8) VOC 2007 (即 Pascal Voc 2007) 是一个关于物体和场景的图像数据集 (如图 1.16 所示), 可以用于目标检测和语义分割任务。Pascal VOC 的全名是 “Pattern Analysis, Statistical Modelling and Computational Learning Visual Object Classes”, 指的是模式分析、统计建模、计算学习视觉物体分类。该数据集包含训练验证集文件夹 trainval 和测试集文件夹 test, 分别包含 5011 幅和 4952 幅大小不同的图像。而且, 这两个文件夹各自又都包含 5 个子文件夹: JPEGImages、Annotations、ImageSets、SegmentationClass 和 SegmentationObject。其中, JPEGImages 存放的是所有图像, 包含 20 个类别。Annotations 存放的是 xml 格式的标签文件, 每一个 xml 文件都对应于 JPEGImages 下的一幅图像。ImageSets 存放的是具体的图像信息, 下设 3 个子文件夹 Layout、Main 和 Segmentation, 分别存放人体部位数据 (比如 head、hand、feet 等)、20 类图像物体识别数据, 以及可用于分割的数据。SegmentationClass 和 SegmentationObject 用来存放分割后的图像, 前者标注每个像素的类别, 后者标注每个像素属于哪一个物体对象。本书在第 8 章的 Faster R-CNN、YOLO 和 SSD 的目标检测案例中使用了 VOC 2007。

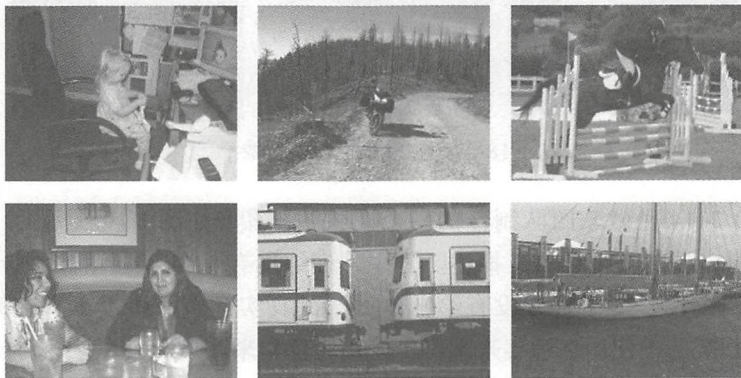


图 1.16 VOC 2007 的图像举例

9) SIFT Flow 是一个关于不同场景的图像数据集 (如图 1.17 所示), 包含 2688 幅图像, 其中 2488 幅训练图像、200 幅测试图像。这些图像的像素共有 33 个语义类别标记 (桥、山、太阳等) 和 3 个几何类别标记 (水平、竖直和天空)。本书在第 9 章的 FCN 图像分割案例中使用了 SIFT Flow。

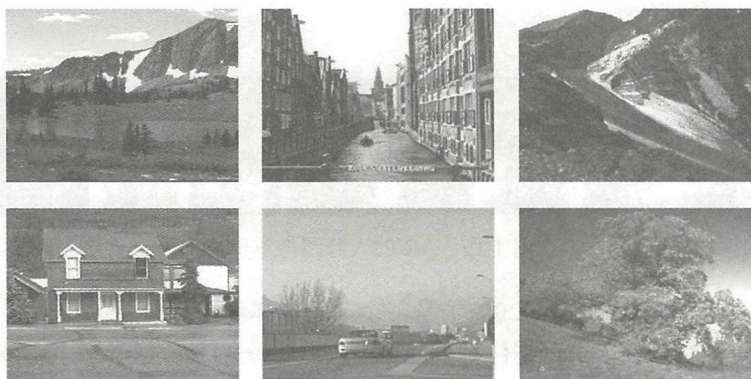


图 1.17 SIFT Flow 的图像举例

10) ADE20K 是一个关于不同场景的图像数据集 (如图 1.18 所示), 包含 20 210 幅训练图像和 2000 幅测试图像。这些图像的像素被标记为 3148 个不同的语义类别。本书在第 9 章的 PSPNet 图像分割案例中使用了 ADE20K。

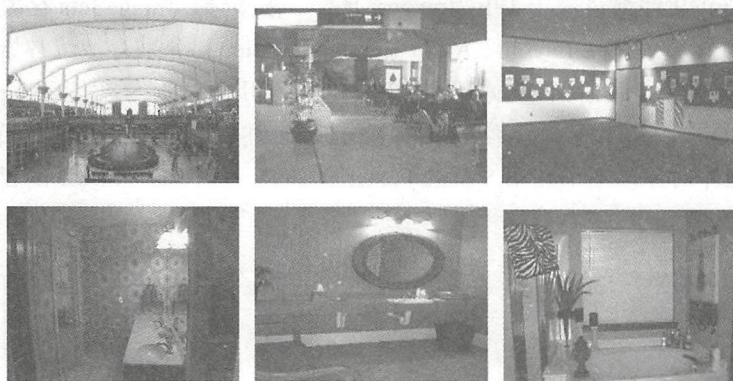


图 1.18 ADE20K 的场景图像举例

11) COCO 2014 是微软团队制作的一个图像数据集 (如图 1.19 所示), 可以用于物体识别、目标检测、语义分割和语义描述。其中, 训练集有 82 783 幅图像, 验证集和测试集分别有 40 504 幅图像。由于官方没有给出测试集的真实标签, 所以本书在第 9 章的 Mask R-CNN 图像分割案例中使用训练集来学习, 使用验证集来测试。

12) CelebA 是一个名人人脸图像数据集 (如图 1.20 所示), 共包含 10 177 位名人的 202 599 幅人脸图像, 其中训练集有 162 770 幅图像, 验证集有 19 867 幅图像, 测试集有 18 962 幅图像。每幅人脸图像有 40 个不同的二值属性标注, 比如是否微笑、是否戴眼镜、是否戴帽子等。本书在第 10 章的 DCGAN 人脸生成案例中使用了 CelebA。

13) Gamerecords 是一个围棋棋局文件数据集 (如图 1.21 所示), 包含从 2001 年开始一直到 2017 年最新更新的棋局文件。本书在第 12 章 AlphaGo 的仿效围棋程序 MuGo 案例中,

使用了2017年4月及之前发布的部分棋局文件,约22 959个。每个棋局文件大概包含200个着子位置,总共约有4 591 800个着子位置,其中4 491 800个着子位置被用来训练,其余的100 000个着子位置用来测试。



图 1.19 COCO 2014 的物体图像举例



图 1.20 CelebA 的人脸图像举例

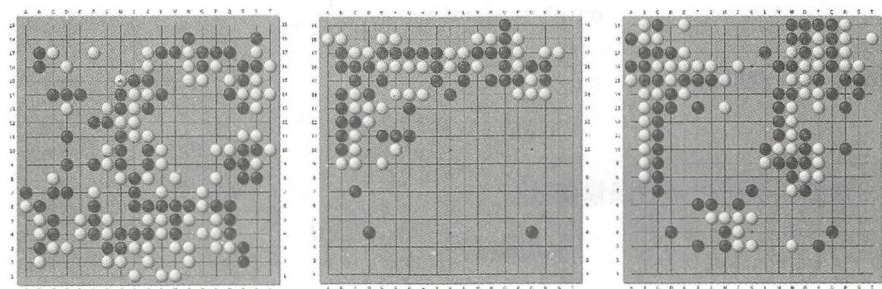


图 1.21 Gamerecords 的棋局举例

第2章

预备知识

本章主要介绍深入理解卷积神经网络涉及的预备知识，包括激活函数、矩阵运算、导数公式、梯度下降算法、反向传播算法（分为通用反向传播算法和逐层反向传播算法）、通用逼近定理、内外卷积运算、膨胀卷积运算、上下采样运算、卷积面计算、池化面计算、局部响应归一化、权值偏置初始化、丢失输出、丢失连接、随机梯度下降算法、块归一化、动态规划算法等。

2.1 激活函数

在神经网络中，虽然理论上激活函数可以是线性的，比如恒等函数 $f(x) = x$ ，但一般选为非线性的 sigmoid 函数：

$$\sigma(x) = \text{sigm}(x) = \frac{1}{1 + e^{-x}} \quad (2.1)$$

或者双曲正切函数 \tanh ：

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2.2)$$

此外，激活函数可以是硬限幅函数：

$$\text{hardlim}(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases} \quad (2.3)$$

或者斜面函数：

$$\text{ramp}(x) = \begin{cases} 1, & x \geq 1 \\ x, & -1 < x < 1 \\ -1, & x \leq -1 \end{cases} \quad (2.4)$$

当然，激活函数还有很多其他选择，比如，

校正线性单元（或修正线性单元）ReLU：

$$\text{ReLU}(x) = \max(0, x) \quad (2.5)$$

渗漏校正线性单元(或渗漏修正线性单元) LReLU:

$$\text{LReLU}(x) = \begin{cases} 1, & x \geq 0 \\ ax, & x < 0 \end{cases} \quad (2.6)$$

其中, $a \in (0, 1)$ 是一个固定值。如果按某个均匀分布取随机值, 则称为 RReLU (Randomized LReLU)。

参数校正线性单元(或参数修正线性单元) PReLU:

$$\text{PReLU}(x) = \begin{cases} 1, & x \geq 0 \\ ax, & x < 0 \end{cases} \quad (2.7)$$

其中, $a \leq 1$ 是一个可调参数, 具体值需要通过学习得到。

指数线性单元 ELU (其中 $a \geq 0$ 是一个可调参数):

$$\text{ELU}(x) = \begin{cases} 1, & x \geq 0 \\ a(e^x - 1), & x < 0 \end{cases} \quad (2.8)$$

软加函数 softplus:

$$f(x) = \ln(1 + e^x) \quad (2.9)$$

最大输出函数 maxout:

$$\text{maxout}(x) = \max(x_1, x_2, \dots, x_n) \quad (2.10)$$

软最大输出函数 softmax:

$$\text{softmax}(x_1, x_2, \dots, x_n) = \frac{1}{\sum_{i=1}^n \exp(x_i)} (\exp(x_i))_{n \times 1} \quad (2.11)$$

2.2 矩阵运算

如果矩阵 $A = (a_{ij})_{m \times n}$, 其转置矩阵 $B = (b_{ij}) = A^T$ 的所有元素定义为

$$b_{ij} = a_{ji}, \forall 1 \leq i \leq n, 1 \leq j \leq m \quad (2.12)$$

如果矩阵 $A = (a_{ij})_{m \times n}$, 其 180° 旋转定义为

$$\text{rot } 180(A) = (a_{m-i+1, n-j+1})_{m \times n} \quad (2.13)$$

给定两个矩阵 $A = (a_{ij})_{m \times n}$ 和 $B = (b_{ij})_{n \times p}$, 它们的乘积 $C = (c_{ij})_{m \times p} = A \cdot B = AB$ 的所有元素定义为

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}, \forall 1 \leq i \leq m, 1 \leq j \leq p \quad (2.14)$$

给定两个矩阵 $A = (a_{ij})_{m \times n}$ 和 $B = (b_{ij})_{m \times n}$, 它们的加法和减法定义为

$$A \pm B = (a_{ij} \pm b_{ij})_{m \times n} \quad (2.15)$$

它们的阿达马积 (Hadamard product), 又称为逐元素积 (elementwise product) 定义为

$$A \circ B = (a_{ij} \cdot b_{ij})_{m \times n} \quad (2.16)$$

给定两个矩阵 $A = (a_{ij})_{m \times n}$ 和 $B = (b_{ij})_{p \times q}$ ，它们的克罗内克积 (Kronnecker product) 定义为

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix} \quad (2.17)$$

如果 $\mathbf{x} = (x_1, x_2, \cdots, x_n)^T$ 是向量，那么一元函数 $f(x)$ 的逐元向量函数 (elementwise vector function) 定义为

$$f(\mathbf{x}) = (f(x_1), f(x_2), \cdots, f(x_n))^T \quad (2.18)$$

如果 $X = (x_{ij})_{m \times n}$ 是矩阵，那么一元函数 $f(x)$ 的逐元矩阵函数 (elementwise matrix function) 定义为

$$f(X) = (f(x_{ij}))_{m \times n} \quad (2.19)$$

逐元向量函数和逐元矩阵函数统称为逐元函数 (elementwise function)。

2.3 导数公式

sigmoid 函数的导数是：

$$\sigma'(x) = (\text{sigm}(x))' = \sigma(x)(1 - \sigma(x)) = -\frac{e^{-x}}{(1 + e^{-x})^2} \quad (2.20)$$

双曲正切函数 \tanh 的导数是：

$$(\tanh(x))' = 1 - (\tanh(x))^2 = \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} \quad (2.21)$$

校正线性单元 ReLU 的导数是：

$$(\text{ReLU}(x))' = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases} \quad (2.22)$$

如果 $\mathbf{x} = (x_1, x_2, \cdots, x_n)^T$ ，那么逐元向量函数的导数是：

$$f'(\mathbf{x}) = (f'(x_1), f'(x_2), \cdots, f'(x_n))^T \quad (2.23)$$

如果 $X = (x_{ij})_{m \times n}$ ，那么逐元矩阵函数的导数是：

$$f'(X) = (f'(x_{ij})) \quad (2.24)$$

如果 \mathbf{a} 、 \mathbf{b} 、 \mathbf{c} 和 \mathbf{x} 是 n 维向量， \mathbf{A} 、 \mathbf{B} 、 \mathbf{C} 和 \mathbf{X} 是 n 阶矩阵，那么

$$\frac{\partial(\mathbf{a}^T \mathbf{x})}{\partial \mathbf{x}} = \frac{\partial(\mathbf{x}^T \mathbf{a})}{\partial \mathbf{x}} = \mathbf{a} \quad (2.25)$$

$$\frac{\partial(\mathbf{a}^T \mathbf{X} \mathbf{b})}{\partial \mathbf{X}} = \mathbf{a} \mathbf{b}^T \quad (2.26)$$

$$\frac{\partial(\mathbf{a}^T \mathbf{X} \mathbf{a})}{\partial \mathbf{X}} = \frac{\partial(\mathbf{a}^T \mathbf{X}^T \mathbf{a})}{\partial \mathbf{X}} = \mathbf{a} \mathbf{a}^T \quad (2.27)$$

$$\frac{\partial(\mathbf{a}^T \mathbf{X}^T \mathbf{X} \mathbf{b})}{\partial \mathbf{X}} = \mathbf{X}(\mathbf{a} \mathbf{b}^T + \mathbf{b} \mathbf{a}^T) \quad (2.28)$$

$$\frac{\partial [(Ax + a)^T C(Bx + b)]}{\partial x} = A^T C(Bx + b) + B^T C(Ax + a) \quad (2.29)$$

$$\frac{\partial (x^T A x)}{\partial x} = (A + A^T)x \quad (2.30)$$

$$\frac{\partial [(Xb + c)^T A(Xb + c)]}{\partial X} = (A + A^T)(Xb + c)b^T \quad (2.31)$$

$$\frac{\partial [b^T X^T A X c]}{\partial X} = A^T X b c^T + A X c b^T \quad (2.32)$$

如果用 $\text{Tr}(\cdot)$ 表示矩阵的迹函数 (即计算矩阵的对角元素之和), 那么不难得到:

$$\frac{\partial [\text{Tr}(f(X))]}{\partial X} = (f'(X))^T \quad (2.33)$$

$$\frac{\partial [\text{Tr}(\sin(X))]}{\partial X} = (\cos(X))^T \quad (2.34)$$

如果 $U = F(X)$ 是关于 X 的矩阵值函数且 $g(U)$ 是关于 U 的实值函数, 那么下面的链式法则成立:

$$\frac{\partial g(U)}{\partial X} = \left(\frac{\partial g(U)}{\partial x_{ij}} \right) = \frac{\partial g(U)}{\partial U} \cdot \frac{\partial U}{\partial X} = \left(\sum_k \sum_l \frac{\partial g(U)}{\partial u_{kl}} \frac{\partial u_{kl}}{\partial x_{ij}} \right) = \left(\text{Tr} \left[\left(\frac{\partial g(U)}{\partial U} \right)^T \frac{\partial U}{\partial x_{ij}} \right] \right) \quad (2.35)$$

此外, 关于矩阵迹函数 $\text{Tr}(\cdot)$ 还有如下偏导公式

$$\frac{\partial [\text{Tr}(AXB)]}{\partial X} = A^T B^T \quad (2.36)$$

$$\frac{\partial [\text{Tr}(AX^T B)]}{\partial X} = BA \quad (2.37)$$

$$\frac{\partial [\text{Tr}(A \otimes X)]}{\partial X} = \text{Tr}(A)I \quad (2.38)$$

$$\frac{\partial [\text{Tr}(AXBX)]}{\partial X} = A^T X^T B^T + B^T X A^T \quad (2.39)$$

$$\frac{\partial [\text{Tr}(C^T X^T BXC)]}{\partial X} = (B^T + B)XCC^T \quad (2.40)$$

$$\frac{\partial [\text{Tr}(X^T BXC)]}{\partial X} = BXC + B^T X C^T \quad (2.41)$$

$$\frac{\partial [\text{Tr}(AXBX^T C)]}{\partial X} = A^T C^T X B^T + C A X B \quad (2.42)$$

$$\frac{\partial [\text{Tr}((AXB + C)(AXB + C)^T)]}{\partial X} = 2A^T (AXB + C)B^T \quad (2.43)$$

2.4 梯度下降算法

梯度下降算法, 又称为最速下降算法, 是在无约束条件下计算连续可微函数极小值的基本方法。这种方法的核心思想是用负梯度方向作为下降方向, 在 1874 年由法国科学家

Cauchy 提出。

设 $f(\mathbf{x})$ 在 \mathbf{x}_k 附近连续可微，令 $\mathbf{x} = \mathbf{x}_k + \alpha \mathbf{d}$ ，其中 \mathbf{d} 为单位方向 ($\|\mathbf{d}\| = 1$)。如果 $\mathbf{g}_k = \nabla f(\mathbf{x}_k) \neq 0$ ，则由 Taylor 展开式得

$$f(\mathbf{x}) = f(\mathbf{x}_k) + (\nabla f(\mathbf{x}_k))^T (\mathbf{x} - \mathbf{x}_k) + o(\|\mathbf{x} - \mathbf{x}_k\|) \quad (2.44)$$

或改写为

$$f(\mathbf{x}_k + \alpha \mathbf{d}) = f(\mathbf{x}_k) + \alpha \mathbf{g}_k^T \mathbf{d} + o(\alpha), \quad \alpha > 0 \quad (2.45)$$

设 θ 为 \mathbf{d} 与 $-\mathbf{g}_k$ 之间的夹角，则有

$$\mathbf{g}_k^T \mathbf{d} = -\|\mathbf{g}_k\| \cos \theta \quad (2.46)$$

不难看出，当 $\theta = 0$ 时， $\cos \theta = 1$ ， $\mathbf{g}_k^T \mathbf{d}$ 取最小值，从而 $f(\mathbf{x})$ 下降最快，此时 $\mathbf{d} = -\mathbf{g}_k$ 。因此，负梯度方向 $-\mathbf{g}_k$ 就是函数 $f(\mathbf{x})$ 在 \mathbf{x}_k 附近下降最快的方向。

根据上述分析，可以把计算函数 $f(\mathbf{x})$ 的梯度下降算法总结为算法 2.1。梯度下降算法一般都是线性收敛的，速度通常较慢。关于梯度下降算法的收敛性可参见文献 [100]。

算法 2.1 梯度下降算法

输入: $f(\mathbf{x})$ 的表达式

输出: 极小值点 \mathbf{x}^*

1. 选初始点 \mathbf{x}_0 ，收敛误差 $\varepsilon > 0$ ，迭代次数为 N ，令 $k=0$ 。
2. 若 $\|\mathbf{g}_k\| \leq \varepsilon$ ，则 $\mathbf{x}^* = \mathbf{x}_k$ ，停止迭代；否则计算 $\mathbf{d}_k = -\mathbf{g}_k$ 。
3. 选择和计算步长因子 α_k 。
4. 计算 $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$ 。
5. 令 $k = k + 1$ ，若 $k \geq N$ ，则 $\mathbf{x}^* = \mathbf{x}_k$ ，停止迭代；否则转步到 2。

如果 $f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{G} \mathbf{x}$ ，其中 \mathbf{G} 是 $n \times n$ 对称正定矩阵，最大和最小特征值分别是 λ_1 和 λ_n ，那么梯度下降算法的收敛速度至少是线性的，且产生的点列 $\{\mathbf{x}_k\}$ 对所有 k 满足

$$\frac{f(\mathbf{x}_{k+1}) - f(\mathbf{x}^*)}{f(\mathbf{x}_k) - f(\mathbf{x}^*)} \leq \frac{(\lambda_1 - \lambda_n)^2}{(\lambda_1 + \lambda_n)^2} \quad (2.47)$$

$$\frac{\|\mathbf{x}_{k+1} - \mathbf{x}^*\|}{\|\mathbf{x}_k - \mathbf{x}^*\|} \leq \sqrt{\frac{\lambda_n}{\lambda_1}} \frac{(\lambda_1 - \lambda_n)}{(\lambda_1 + \lambda_n)} \quad (2.48)$$

其中， \mathbf{x}^* 是问题的唯一极小点。

2.5 反向传播算法

卷积神经网络是一种特殊的前馈神经网络，通常也是一种深度神经网络。深度学习则是在克服反向传播算法对深度神经网络的训练困难过程中逐步发展和建立起来的。而深度神

神经网络的基本学习训练方法是反向传播算法。因此,有必要提前介绍一下反向传播算法的具体内容。作为一种有监督学习算法,反向传播算法在本质上是一种具有递归结构的梯度下降算法,往往需要给定足够多的训练样本,才能获得满意的效果。下面先给出任意前馈神经网络的通用反向传播算法,再讨论深层神经网络的逐层反向传播算法。这两个反向传播算法是读者理解其他各种神经网络的反向传播算法的基础。

2.5.1 通用反向传播算法

不妨设前馈神经网络共包含 N 个节点 $\{u_1, u_2, \dots, u_N\}$, 只有从编号较小的神经元才能连接到编号较大的神经元, 没有反馈连接。图 2.1 给出了前馈神经网络的一种可能的连接结构, 共包含 10 个节点, 其中两个为输入。一般地, 用 $x_{n,l}$ 表示第 n 个节点对第 l 个输入样本的输出, 其中 $1 \leq n \leq N$ 且 $1 \leq l \leq L$ 。

如果 u_n 是输入节点, 那么它对第 l 个输入样本的输入为 $\text{net}_{n,l} = x_{n,l}$; 否则, u_n 是隐含节点或输出节点, 相应的输入为 $\text{net}_{n,l} = \sum_k w_{k \rightarrow n} x_{k,l}$, 输出为 $x_{n,l} = f_n(\text{net}_{n,l})$ 。其中, $w_{k \rightarrow n} (k < n)$ 表示从第 k 个节点到第 n 个节点的有向连接 $k \rightarrow n$ 的权值, f_n 表示第 n 个节点的激活函数, 比如 sigmoid 函数。此外, 若令 $x_{0,l} = 1$, 则可用 $w_{0 \rightarrow n}$ 表示非输入节点 u_n 的偏置值。最后, 用 OUT 表示所有输出神经元的集合, 且对任意 $n \in \text{OUT}$, 用 $y_{n,l}$ 表示 $x_{n,l}$ 的期望值, 用 $e_{n,l}$ 表示编号为 n 的输出神经元对第 l 个输入样本产生的输出误差。因此, 关于样本 l 的输出误差可以表示为

$$E_l = \sum_{n \in \text{OUT}} e_{n,l} \quad (2.49)$$

总的输出误差为

$$E = \sum_{l=1}^L E_l = \sum_{l=1}^L \sum_{n \in \text{OUT}} e_{n,l} \quad (2.50)$$

如果定义第 n 个神经元关于第 l 个样本的反传误差信号 (backpropagated error signal) 或灵敏度 (sensitivity) 如下:

$$\delta_{n,l} = \frac{\partial E}{\partial \text{net}_{n,l}} = \frac{\partial E_l}{\partial \text{net}_{n,l}} \quad (2.51)$$

那么利用链式法不难得到:

$$\forall n \in \text{OUT}, \frac{\partial E}{\partial w_{k \rightarrow n}} = \sum_{l=1}^L \frac{\partial E_l}{\partial \text{net}_{n,l}} \frac{\partial \text{net}_{n,l}}{\partial w_{k \rightarrow n}} = \sum_{l=1}^L \frac{\partial E_l}{\partial x_{n,l}} f'_n(\text{net}_{n,l}) \frac{\partial \text{net}_{n,l}}{\partial w_{k \rightarrow n}} = \sum_{l=1}^L \delta_{n,l} \frac{\partial \text{net}_{n,l}}{\partial w_{k \rightarrow n}} = \sum_{l=1}^L \delta_{n,l} x_{k,l} \quad (2.52)$$

$$\begin{aligned} \forall k \notin \text{OUT}, \frac{\partial E}{\partial w_{j \rightarrow k}} &= \sum_{l=1}^L \frac{\partial E_l}{\partial \text{net}_{k,l}} \frac{\partial \text{net}_{k,l}}{\partial w_{j \rightarrow k}} = \sum_{l=1}^L \left(\sum_{n \in \text{OUT}} \frac{\partial E_l}{\partial \text{net}_{n,l}} \frac{\partial \text{net}_{n,l}}{\partial \text{net}_{k,l}} \right) \frac{\partial \text{net}_{k,l}}{\partial w_{j \rightarrow k}} \\ &= \sum_{l=1}^L \left(\sum_{n \in \text{OUT}} w_{k \rightarrow n} \delta_{n,l} \right) f'_k(\text{net}_{k,l}) \frac{\partial \text{net}_{k,l}}{\partial w_{j \rightarrow k}} = \sum_{l=1}^L \delta_{k,l} \frac{\partial \text{net}_{k,l}}{\partial w_{j \rightarrow k}} = \sum_{l=1}^L \delta_{k,l} x_{j,l} \end{aligned} \quad (2.53)$$

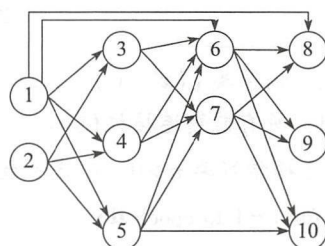


图 2.1 前馈神经网络的示意图, 共包含 10 个节点, 其中节点 1 和 2 为输入, 节点 8、9 和 10 为输出

如果对输出节点选择平方误差 $e_{n,l} = \frac{1}{2} (x_{n,l} - y_{n,l})^2$, 那么

$$\forall n \in \text{OUT}, \delta_{n,l} = \frac{\partial e_{n,l}}{\partial \text{net}_{n,l}} = (x_{n,l} - y_{n,l}) f'_n(\text{net}_{n,l}) \quad (2.54)$$

$$\forall n \notin \text{OUT}, \delta_{k,l} = \left(\sum_{k \rightarrow n} w_{k \rightarrow n} \delta_{n,l} \right) f'_k(\text{net}_{k,l}) \quad (2.55)$$

因此, 相应的通用反向误差传播算法可以总结为算法 2.2。

注意, 在算法 2.2 中, epoch 表示迭代次数, 在实际应用中可能需要几十次、几百次, 甚至成千上万次迭代, 才能获得令人满意的学习训练效果。

算法 2.2 通用反向传播算法

输入: 训练集 $\{(x^l, y^l), 1 \leq l \leq L\}$, 学习率 η , 前馈网络结构, 迭代次数 epoch

输出: 所有前馈连接权值 $w_{k \rightarrow n}$

1. 选择学习率 $\eta > 0$, 迭代次数 epoch, 随机初始化 $w_{k \rightarrow n}$
2. **for** $i = 1$ to epoch **do**
3. **for** $l = 1$ to L **do**
4. 如果 n 是输出节点, 则计算其反传误差信号 $\delta_{n,l} = (x_{n,l} - y_{n,l}) f'_n(\text{net}_{n,l})$
5. 否则递归计算其反传误差信号 $\delta_{k,l} = \left(\sum_{k \rightarrow n} w_{k \rightarrow n} \delta_{n,l} \right) f'_k(\text{net}_{k,l})$
6. **end for**
7. 计算关于连接权的偏导 $\Delta w_{k \rightarrow n} = \frac{\partial E}{\partial w_{k \rightarrow n}} = \sum_{l=1}^L \delta_{n,l} x_{k,l}$
8. 如果所有这些偏导产生的总梯度足够小, 则停止
9. 否则, 更新连接权值 $w_{k \rightarrow n} = w_{k \rightarrow n} - \eta \Delta w_{k \rightarrow n}$
10. **end for**

2.5.2 逐层反向传播算法

虽然通用反向传播算法在理论上适用于任何前馈神经网络, 但是对于多层感知器来说, 还可以设计出更加简洁实用的逐层反向传播算法^[10]。

多层感知器由一个输入层、若干隐含层和一个输出层组成, 如图 2.2 所示。不妨设输入层的维数是 m , 输入向量表示为 $x = (x_1, x_2, \dots, x_m)^T \in R^m$ 。第 k 个隐含层包含 n_k 个神经元 ($k = 1, 2, \dots, R$), 相应的隐含层向量表示为 $h_k = (h_{k,1}, h_{k,2}, \dots, h_{k,n_k})^T \in R^{n_k}$ 。输出层包含 c 个神经元, 输出向量表示为 $o = (o_1, o_2, \dots, o_c)^T \in R^c$ 。输入层与第 1 个隐含层之间的权值矩阵用

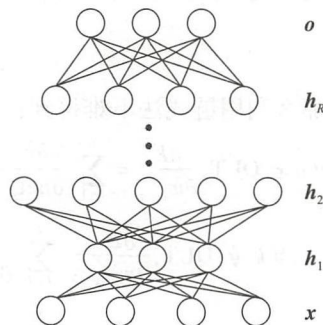


图 2.2 多层感知器的示意图

$W^1 = (w_{ij}^1)_{n_1 \times m}$ 表示, 第 $k-1$ 个隐含层与第 k 个隐含层之间的权值矩阵用 $W^k = (w_{ij}^k)_{n_k \times n_{k-1}}$ ($1 < k \leq R$) 表示, 第 R 个隐含层与输出层之间的权值矩阵用 $W^{R+1} = (w_{ij}^{R+1})_{c \times n_R}$ 表示。那么, 这个多层感知器的各层神经元激活输出可以计算如下:

$$\begin{cases} h_1 = \sigma_{h_1}(W^1 x + b^1) \\ h_k = \sigma_{h_k}(W^k h_{k-1} + b^k), 2 \leq k \leq R \\ o = \sigma_o(W^{R+1} h_R + b^{R+1}) \end{cases} \quad (2.56)$$

其中, b^1 、 b^k 和 b^{R+1} 是各层的偏置; σ_{h_1} 、 σ_{h_k} 和 σ_o 是各层的逐元向量函数, 一般都取为逐元 sigmoid 函数。

算法 2.3 逐层反向传播算法

输入: 训练集 $\{(x^l, y^l), 1 \leq l \leq L\}$, 学习率 η , 分层网络结构, 隐含层数 R , 迭代次数 epoch

输出: 所有权值和偏置 $(W^k, b^k) (1 \leq k \leq R)$

1. 随机初始化 $W^k \approx 0, b^k \approx 0, k = 1, \dots, R$
2. 令 $h_0^l = x^l$
3. for $i = 1, 2, \dots, \text{epoch}$ do
4. for $l = 1, 2, \dots, L$ do
5. 计算 $u_k^l = W^k h_{k-1}^l + b^k, h_k^l = \sigma(u_k^l), 1 \leq k \leq R+1$
6. 计算 $\delta_{R+1}^l = (o^l - y^l) \cdot \sigma'(u_{R+1}^l)$
7. 计算 $\delta_k^l = [(W^{k+1})^T \delta_{k+1}^l] \cdot \sigma'(u_k^l), 1 \leq k \leq R$
8. end for
9. 计算 $\frac{\partial E}{\partial W^k} = \sum_{l=1}^L \delta_k^l (h_{k-1}^l)^T, \frac{\partial E}{\partial b^k} = \sum_{l=1}^L \delta_k^l, 1 \leq k \leq R+1$
10. 如果总梯度足够小, 则停止
11. 否则, 更新权值和偏置: $W^k \leftarrow W^k - \eta \frac{\partial L_N}{\partial W^k}, b^k \leftarrow b^k - \eta \frac{\partial L_N}{\partial b^k}, 1 \leq k \leq R+1$
12. end for

为了对多层感知器中的权值和偏置进行学习, 需要给定一组训练样本。假定共有 L 个训练样本 $(x^l, y^l) (1 \leq l \leq L)$, 输入是 $x^l = (x_1^l, x_2^l, \dots, x_m^l)^T$, 期望输出是 $y^l = (y_1^l, y_2^l, \dots, y_c^l)^T$, 实际输出是 $o^l = (o_1^l, o_2^l, \dots, o_c^l)^T$ 。把优化的目标函数选为平方误差:

$$E = \frac{1}{2} \sum_{l=1}^L \|o^l - y^l\|^2 = \frac{1}{2} \sum_{l=1}^L \sum_{j=1}^c (o_j^l - y_j^l)^2 \quad (2.57)$$

令 $h_0^l = x^l, u_k^l = W^k h_{k-1}^l + b^k, h_k^l = \sigma(u_k^l) (1 \leq k \leq R+1), o^l = h_{R+1}^l = \sigma(u_{R+1}^l)$ 。如果定义网络各层关于第 l 个样本的反传误差信号或灵敏度为

$$\delta_k^l = \frac{\partial E}{\partial \mathbf{u}_k^l} \quad (2.58)$$

那么这些反传误差信号可以递归计算如下：

$$\delta_{R+1}^l = \frac{\partial E}{\partial \mathbf{u}_{R+1}^l} = \frac{\partial E}{\partial \mathbf{o}^l} \cdot \frac{\partial \mathbf{o}^l}{\partial \mathbf{u}_{R+1}^l} = (\mathbf{o}^l - \mathbf{y}^l) \circ \sigma'(\mathbf{u}_{R+1}^l) = (\mathbf{o}^l - \mathbf{y}^l) \circ \mathbf{o}^l \circ (1 - \mathbf{o}^l) \quad (2.59)$$

$$\delta_k^l = \frac{\partial E}{\partial \mathbf{u}_k^l} = \frac{\partial E}{\partial \mathbf{u}_{k+1}^l} \cdot \frac{\partial \mathbf{u}_{k+1}^l}{\partial \mathbf{u}_k^l} = [(\mathbf{W}^{k+1})^T \delta_{k+1}^l] \circ \sigma'(\mathbf{u}_k^l), 1 \leq k \leq R \quad (2.60)$$

综上所述，逐层反向传播算法可以总结为算法 2.3。另外，如果采用受限波耳兹曼机对多层感知器的权值和偏置进行预训练代替随机初始化，还可以进一步提高逐层反向传播算法的学习训练效果^[32]。

此外，学习训练过程优化的目标函数不一定是平方误差，还可以是其他函数，比如交叉熵：

$$E = - \sum_{l=1}^L \sum_{j=1}^c (y_j^l \times \log(o_j^l) + (1 - y_j^l) \times \log(1 - o_j^l)) \quad (2.61)$$

这时，相应的逐层反向传播算法只需把 δ_{R+1}^l 的计算公式修改为

$$\delta_{R+1}^l = \mathbf{o}^l - \mathbf{y}^l \quad (2.62)$$

最后，需要指出的是，多层感知器在训练完成之后，常常再被用软最大函数转换成伪概率输出。虽然很多文献和代码都把这种软最大函数转换称为软最大输出，但从理论上严格地说，软最大输出应该是把输出层的计算定义为

$$\mathbf{o} = \text{softmax}(\mathbf{W}^{R+1} \mathbf{h}_R + \mathbf{b}^{R+1}) \quad (2.63)$$

如果保持其他部分不变，但采用式 (2.63) 计算网络的输出，那么在采用平方误差 (2.57) 作为目标函数时， δ_{R+1}^l 的计算公式应改为

$$\delta_{R+1}^l = \frac{\partial E}{\partial \mathbf{u}_{R+1}^l} = \frac{\partial E}{\partial \mathbf{o}^l} \cdot \frac{\partial \mathbf{o}^l}{\partial \mathbf{u}_{R+1}^l} = [\text{diag}(\mathbf{o}^l) - \mathbf{o}^l (\mathbf{o}^l)^T] (\mathbf{o}^l - \mathbf{y}^l) \quad (2.64)$$

而在采用交叉熵 (2.61) 作为目标函数时， δ_{R+1}^l 的计算公式应改为

$$\begin{aligned} \delta_{R+1}^l &= \begin{pmatrix} 1 & \frac{o_1^l}{o_2^l - 1} & \cdots & \frac{o_1^l}{o_c^l - 1} \\ \frac{o_2^l}{o_1^l - 1} & 1 & \cdots & \frac{o_2^l}{o_c^l - 1} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{o_c^l}{o_1^l - 1} & \frac{o_c^l}{o_2^l - 1} & \cdots & 1 \end{pmatrix} (\mathbf{o}^l - \mathbf{y}^l) \\ &= (\text{diag}(\mathbf{1} ./ (\mathbf{1} - \mathbf{o}^l)) - (\mathbf{o}^l)^T (\mathbf{1} ./ (\mathbf{1} - \mathbf{o}^l))) (\mathbf{o}^l - \mathbf{y}^l) \end{aligned} \quad (2.65)$$

其中，“1”表示一个分量全是1的向量，“./”表示向量的对应分量相除。

如果每个样本的期望输出 y^l 仅有一个分量, 为 $y_{jl}^l = 1$, 那么还可以选用退化交叉熵作为目标函数, 即

$$E = - \sum_{l=1}^L \log(o_{jl}^l) \quad (2.66)$$

同时, δ_{R+1}^l 的计算公式应修正为

$$\delta_{R+1}^l = o^l - y^l \quad (2.67)$$

2.6 通用逼近定理

多层感知器是一种非常著名的人工神经网络模型, 如果包含足够多的隐含神经元, 那么即使只有一个隐含层, 它所表达的输入输出映射在理论上也能够充分逼近任何一个定义在单位立方体上的连续函数。这就是通用逼近定理的核心内容^[22-24], 其严谨的数学描述和表达如下。

令激活函数 φ 是一个非常数、有界且单调递增的连续函数。 $I_m = [0, 1]^m$ 表示 m 维空间的单位超立方体, $C(I_m)$ 表示 I_m 上的连续函数空间。那么, 给定任意连续函数 $f \in C(I_m)$ 和 $\varepsilon > 0$, 存在一个正整数 $n > 0$, 实常数 $\alpha_i, w_{ij}, b_j (1 \leq i \leq n, 1 \leq j \leq m)$, 使得

$$F(x_1, x_2, \dots, x_m) = \sum_{i=1}^n \alpha_i \varphi \left(\sum_{j=1}^m w_{ij} x_j + b_i \right) \quad (2.68)$$

对输入空间中的所有 x_1, x_2, \dots, x_m , 满足

$$|F(x_1, x_2, \dots, x_m) - f(x_1, x_2, \dots, x_m)| < \varepsilon \quad (2.69)$$

2.7 内外卷积运算

在卷积神经网络中, 可能涉及两种卷积运算: 内卷积和外卷积。在目前的科技文献中一般都不把这两种卷积明确区分开来, 这有时可能引起逻辑和理解上的混乱。虽然卷积神经网络在前向计算时只用到内卷积, 但是在设计反向传播学习算法时则要用到外卷积。下面是内卷积和外卷积的定义。

假设 A 和 B 为矩阵, 大小分别为 $M \times N$ 和 $m \times n$, 且 $M \geq m, N \geq n$, 则它们的内卷积 $C = A * B$ 的所有元素定义为

$$c_{ij} = \sum_{s=1}^m \sum_{t=1}^n a_{i+m-s, j+n-t} \cdot b_{st}, 1 \leq i \leq M-m+1, 1 \leq j \leq N-n+1 \quad (2.70)$$

它们的外卷积定义为

$$A \hat{*} B = \hat{A}_B \tilde{*} B \quad (2.71)$$

其中, $\hat{A}_B = (\hat{a}_{ij})$ 是一个利用 0 对 A 进行扩充得到的矩阵, 大小为 $(M+2m-2) \times (N+2n-2)$, 且

$$\hat{a}_{ij} = \begin{cases} a_{i-m+1, j-n+1}, & m \leq i \leq M+m-1 \text{ 且 } n \leq j \leq N+n-1 \\ 0, & \text{其他} \end{cases} \quad (2.72)$$

例如, 假设矩阵 $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ 和矩阵 $B = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$, 则对 A 和 B 进行内卷积和外卷积

的结果分别为

$$A \tilde{*} B = \begin{bmatrix} 35 & 49 \\ 77 & 91 \end{bmatrix} \quad A \hat{*} B = \begin{bmatrix} 2 & 7 & 12 & 9 \\ 12 & 35 & 49 & 33 \\ 30 & 77 & 91 & 57 \\ 28 & 67 & 76 & 45 \end{bmatrix} \quad (2.73)$$

如果 $C = A \tilde{*} B$ 且 $E(C)$ 是一个关于 C 的任意可微函数, 那么有如下卷积链式法则:

$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial C} \cdot \frac{\partial C}{\partial B} = A \tilde{*} \frac{\partial E}{\partial C} \quad (2.74)$$

$$\frac{\partial E}{\partial A} = \frac{\partial E}{\partial C} \cdot \frac{\partial C}{\partial A} = \frac{\partial E}{\partial C} \hat{*} \text{rot}180(B) \quad (2.75)$$

2.8 膨胀卷积运算

假设 $F: Z^2 \rightarrow R$ 为离散函数, 假设 $\Omega_r = [-r, r]^2 \cap Z^2$, $k: \Omega_r \rightarrow R$ 为大小为 $(2r+1)^2$ 的离散卷积核。 F 和 k 的离散卷积还可定义为

$$(F * k)(p) = \sum_{s+t=p} F(s)k(t) \quad (2.76)$$

膨胀卷积 (dilated convolution), 又称为扩张卷积, 是对上述离散卷积的泛化。假设 l 为膨胀因子 (dilation factor), 则相应的 l -膨胀卷积 $*_l$ 定义为:

$$(F *_l k)(p) = \sum_{s+lt=p} F(s)k(t) \quad (2.77)$$

显然, 1-膨胀卷积就是式 (2.76) 描述的普通离散卷积。膨胀卷积支持以倍数方式扩大感受野。一个用 3×3 卷积核定义的离散卷积, 经过 2 倍因子膨胀后感受野的大小将由原来的 3×3 变成 7×7 , 经过 4 倍因子膨胀后感受野的大小则将变成 15×15 。卷积核和感受野的膨胀过程如图 2.3 所示。

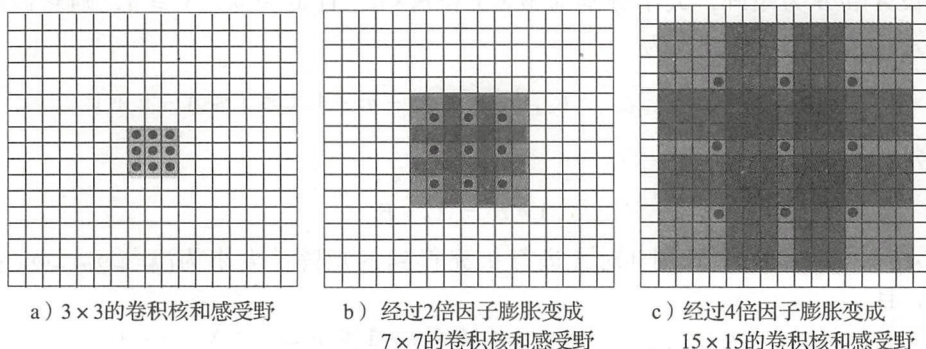


图 2.3 卷积核和感受野的膨胀过程

2.9 上下采样运算

在卷积神经网络中，还可能涉及两种采样运算：上采样和下采样。上采样与下采样之间存在着某种对应关系。不同的下采样运算，相应的上采样运算一般是不同的。常用的下采样有两种：平均下采样（average downsampling，或 mean downsampling）和最大下采样（max downsampling）。这两种下采样又分别称为平均池化和最大池化。相应的上采样称为平均上采样和最大上采样。

对一个矩阵 \mathbf{A} 进行下采样，首先要对它分块。标准的分块操作是不重叠的，理论上分块也可以是重叠的，但分块的数目相对较多。如果分块不重叠且每块的大小为 $\lambda \times \tau$ ，则其中的第 ij 个块可以表示为

$$\mathbf{G}_{\lambda, \tau}^{\mathbf{A}}(i, j) = (a_{st})_{\lambda \times \tau} \quad (2.78)$$

其中， $(i-1) \cdot \lambda + 1 \leq s \leq i \cdot \lambda, (j-1) \cdot \tau + 1 \leq t \leq j \cdot \tau$ 。

对 $\mathbf{G}_{\lambda, \tau}^{\mathbf{A}}(i, j)$ 的平均下采样定义为

$$\text{avgdown}(\mathbf{G}_{\lambda, \tau}^{\mathbf{A}}(i, j)) = \frac{1}{\lambda \times \tau} \sum_{s=(i-1) \times \lambda + 1}^{i \times \lambda} \sum_{t=(j-1) \times \tau + 1}^{j \times \tau} a_{st} \quad (2.79)$$

对 $\mathbf{G}_{\lambda, \tau}^{\mathbf{A}}(i, j)$ 的最大下采样定义为

$$\text{maxdown}(\mathbf{G}_{\lambda, \tau}^{\mathbf{A}}(i, j)) = \max\{a_{st}, (i-1) \cdot \lambda + 1 \leq s \leq i \cdot \lambda, (j-1) \cdot \tau + 1 \leq t \leq j \cdot \tau\} \quad (2.80)$$

如果用大小为 $\lambda \times \tau$ 的块对矩阵 \mathbf{A} 进行不重叠平均下采样，结果定义为

$$\mathbf{D}_{\text{avg}} = \text{avgdown}_{\lambda, \tau}(\mathbf{A}) = (\text{avgdown}(\mathbf{G}_{\lambda, \tau}^{\mathbf{A}}(i, j))) \quad (2.81)$$

相应地，对矩阵 \mathbf{D}_{avg} 进行倍数为 $\lambda \times \tau$ 的不重叠平均上采样，结果定义为

$$\text{avgup}_{\lambda \times \tau}(\mathbf{D}_{\text{avg}}) = \mathbf{D}_{\text{avg}} \otimes \mathbf{1}_{\lambda \times \tau} \quad (2.82)$$

其中， $\mathbf{1}_{\lambda \times \tau}$ 是一个元素全为 1 的矩阵， \otimes 代表克罗内克积。

如果用大小为 $\lambda \times \tau$ 的块对矩阵 \mathbf{A} 进行不重叠最大下采样，结果定义为

$$\mathbf{D}_{\text{max}} = \text{maxdown}_{\lambda, \tau}(\mathbf{A}) = (\text{maxdown}(\mathbf{G}_{\lambda, \tau}^{\mathbf{A}}(i, j))) \quad (2.83)$$

相应地，对矩阵 $\mathbf{D}_{\text{max}} = (d_{ij})$ 进行倍数为 $\lambda \times \tau$ 的不重叠最大上采样，结果定义为一个分块矩阵，即

$$\text{maxup}_{\lambda \times \tau}(\mathbf{D}_{\text{max}}) = (\mathbf{U}_{ij}) \quad (2.84)$$

其中，所有 $\mathbf{U}_{ij} = (u_{kl})_{\lambda \times \mu}$ 都是大小为 $\lambda \times \tau$ 的矩阵，每个元素的定义如下：

$$u_{kl} = \begin{cases} d_{ij}, & k = s^* \text{ 且 } l = t^*, \text{ 其中 } s^* t^* = \arg \max \{a_{st} \in \mathbf{G}_{\lambda, \tau}^{\mathbf{A}}(i, j)\} \\ 0, & \text{其他} \end{cases} \quad (2.85)$$

根据上述定义，如果矩阵 $\mathbf{A} = \begin{pmatrix} 3 & 6 & 8 & 4 \\ 4 & 7 & 7 & 1 \\ 2 & 2 & 4 & 2 \\ 2 & 4 & 3 & 1 \end{pmatrix}$ ，那么其 2×2 不重叠平均下采样和最大下

采样分别为

$$D_{\text{avg}} = \text{avgdown}_{2,2}(A) = \begin{pmatrix} 5 & 5 \\ 2.5 & 2.5 \end{pmatrix}, D_{\text{max}} = \text{maxdown}_{2,2}(A) = \begin{pmatrix} 7 & 8 \\ 4 & 4 \end{pmatrix} \quad (2.86)$$

相应的平均上采样和最大上采样分别为

$$\text{avgup}_{2,2}(D_{\text{avg}}) = \begin{pmatrix} 5 & 5 & 5 & 5 \\ 5 & 5 & 5 & 5 \\ 2.5 & 2.5 & 2.5 & 2.5 \\ 2.5 & 2.5 & 2.5 & 2.5 \end{pmatrix}, \text{maxup}_{2,2}(D_{\text{max}}) = \begin{pmatrix} 0 & 0 & 8 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 4 & 0 & 0 \end{pmatrix} \quad (2.87)$$

2.10 卷积面计算

在卷积神经网络中，一个卷积层可以包含很多卷积面。卷积面又称为卷积特征图（convolutional feature map）或卷积图（convolutional map），有时也称为特征图（feature map）。每个卷积面都是根据输入、卷积核和激活函数来计算的。卷积面的输入通常是一幅或多幅图像。卷积核是一个矩阵（或张量），又称为卷积滤波器，简称滤波器。激活函数有很多不同的选择，但一般选为 sigmoid 函数或校正线性单元（ReLU）。

如果输入是一幅大小为 $M \times N$ 的图像，用矩阵 x 表示，卷积核是大小为 $m \times n$ 的矩阵 w ，偏置为 b ，那么卷积面的计算过程可表示为图 2.4，或者表示为如下公式：

$$h = x * w + b \quad (2.88)$$

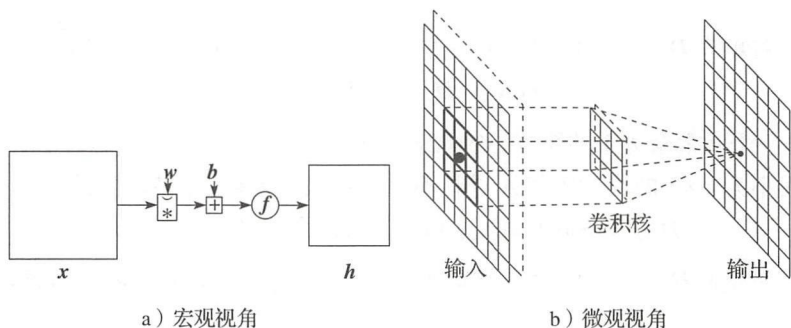


图 2.4 单幅图像输入时卷积面的计算过程

如果输入是 D 个通道，即有 D 幅图像 x_1, x_2, \dots, x_D ，相应的卷积核分别为 w_1, w_2, \dots, w_D ，偏置为 b ，那么这时卷积面的计算过程可表示为图 2.5a，或简化为图 2.5b，或直接用公式表示：

$$h = \sum_{i=1}^D x_i * w_i + b \quad (2.89)$$

此外，在输入多幅图像时，还可采用多组卷积核和偏置，每组产生一个卷积面。如果把这些卷积面按顺序拼接成一个立方体，可以得到多卷积面计算的简化表示，如图 2.5c 所示。

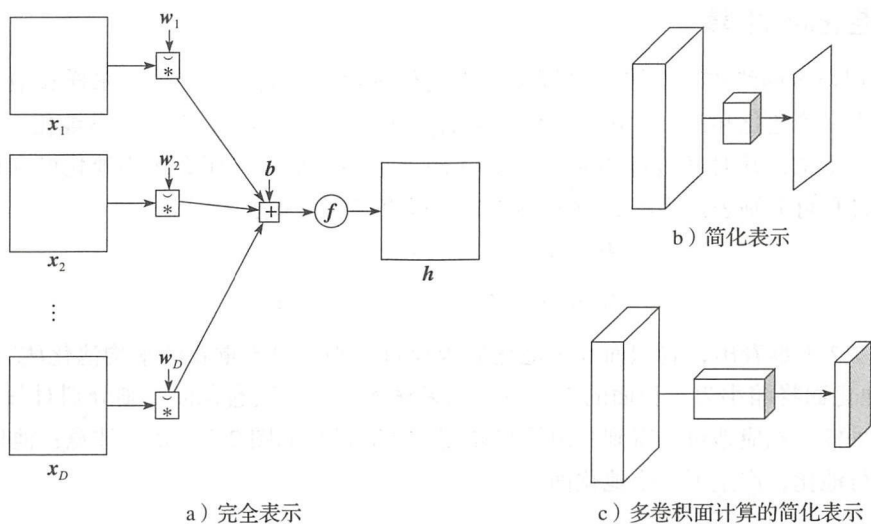


图 2.5 多幅图像输入时卷积面的计算过程

最后，需要指出的是，在现代卷积网络中，卷积面的计算有时也采用非标准卷积代替卷积。非标准卷积是对卷积的推广，其基本思想在于将跨度引入卷积。跨度分为垂直跨度和水平跨度，分别用 u 和 v 表示。假设 A 和 B 为矩阵，大小分别为 $M \times N$ 和 $m \times n$ ，且 $M \geq m$ ， $N \geq n$ ，则它们的非标准内卷积 $C = A \overset{v}{*}_u B$ 的所有元素定义为

$$c_{ij} = \sum_{s=1}^m \sum_{t=1}^n a_{(i-1)u+1+m-s, (j-1)v+1+n-t} \cdot b_{st}, 1 \leq i \leq \frac{M-m}{u} + 1, 1 \leq j \leq \frac{N-n}{v} + 1 \quad (2.90)$$

如果采用非标准卷积，单卷积面和多卷积面计算可以分别简化表示图 2.6a 和图 2.6b。

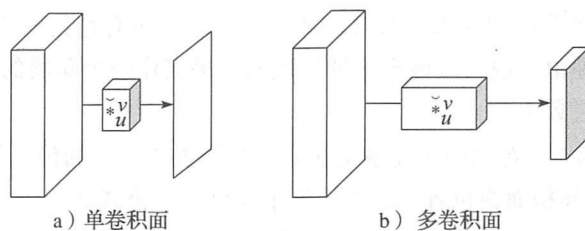


图 2.6 非标准卷积面计算的简化表示

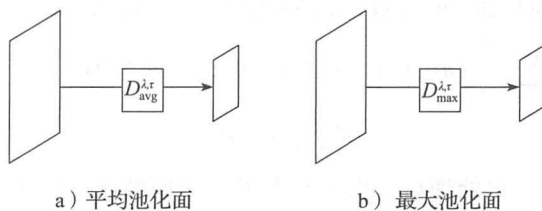


图 2.7 池化面计算的简化表示

2.11 池化面计算

在卷积神经网络中，下采样过程又称为池化过程。相应地，平均下采样和最大下采样又分别称为平均池化和最大池化。池化面的输入既可以是卷积面，也可以是池化面。如果输入的是卷积面 C ，并对其进行块大小为 $\lambda \times \tau$ 的不重叠下采样，那么平均池化面和最大池化面的计算过程可分别表示为图 2.7a 和图 2.7b，或直接用公式表示：

$$H = D_{\text{avg}}^{\lambda, \tau}(C) = \text{avgdown}_{\lambda, \tau}(C) \quad (2.91)$$

$$H = D_{\text{max}}^{\lambda, \tau}(C) = \text{maxdown}_{\lambda, \tau}(C) \quad (2.92)$$

从图 2.7 不难看出，卷积面经过池化后规模将变小。对不重叠的平均池化 $D_{\text{avg}}^{\lambda, \tau}$ 和最大池化 $D_{\text{max}}^{\lambda, \tau}$ ，池化面将缩小为卷积面的 $1/(\lambda\mu)$ 。如果输入的是一组卷积面，则分别对每个面进行不重叠下采样，相应地可以得到一组下采样面，计算过程如图 2.8 所示。注意：池化面 H 也可以再进行池化，产生下一个池化面。

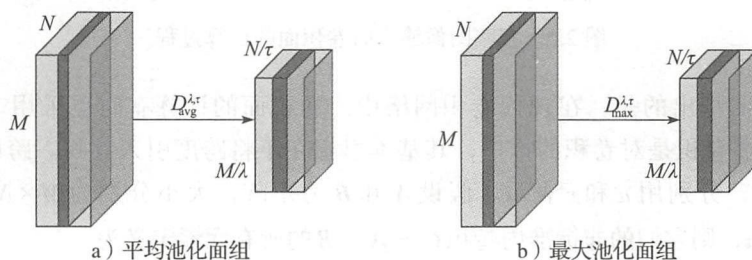


图 2.8 池化面按组计算的简化表示

2.12 局部响应归一化

为了改善卷积神经网络的效果，有时需要对某一层的所有卷积面（或池化面）逐一进行局部归一化处理。从理论上说，应该把这种处理的结果当作一个新增的层来看待，但实际应用时一般不被统计到层数中。

如果用 $a_{x,y}^i$ 表示第 i 个卷积面（或池化面）上在位置 (x, y) 的值，则局部响应归一化的值 $b_{x,y}^i$ 是通过若干相邻卷积面在位置 (x, y) 的值来计算的，公式如下：

$$b_{x,y}^i = a_{x,y}^i / \left(k + \alpha \sum_{j=\max(0, i-n/2)}^{\min(N-1, i+n/2)} (a_{x,y}^j)^2 \right)^\beta \quad (2.93)$$

其中， N 是卷积面（或池化面）的总数， n 是相邻面的个数， k 、 α 、 β 是可调参数。

令 $M_i = \left(k + \alpha \sum_{j=\max(0, i-n/2)}^{\min(N-1, i+n/2)} (a_{x,y}^j)^2 \right)$ ，则式 (2.91) 可改写为

$$b_{x,y}^i = a_{x,y}^i M_i^{-\beta} \quad (2.94)$$

在经过局部响应归一化处理后，由于整个网络的模型已经发生了变化，所以其学习训练过程也需要做相应的调整。这种调整涉及如下偏导数的计算：

$$\begin{aligned}
\frac{\partial E}{\partial a_{x,y}^i} &= \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} \frac{\partial E}{\partial b_{x,y}^j} \frac{\partial b_{x,y}^j}{\partial a_{x,y}^i} = \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} \delta_{x,y}^j \frac{\partial b_{x,y}^j}{\partial a_{x,y}^i} \\
&= \delta_{x,y}^i M_i^{-\beta} - (2\alpha\beta) \cdot a_{x,y}^i \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (\delta_{x,y}^j \cdot b_{x,y}^j \cdot M_j^{-1})
\end{aligned} \quad (2.95)$$

其中, E 为目标函数, $\delta_{x,y}^j = \frac{\partial E}{\partial b_{x,y}^j}$, 且 $\frac{\partial b_{x,y}^j}{\partial a_{x,y}^i}$ 的计算公式如下:

$$\frac{\partial b_{x,y}^j}{\partial a_{x,y}^i} = \begin{cases} M_i^{-\beta} - (2\alpha\beta) \cdot a_{x,y}^i \cdot b_{x,y}^i \cdot M_i^{-1}, & j = i \\ -2\alpha\beta a_{x,y}^i b_{x,y}^j M_j^{-1}, & j \neq i \end{cases} \quad (2.96)$$

2.13 权值偏置初始化

在训练神经网络之前, 必须对其权值和偏置进行初始化。常用的初始化方法有三种, 分别是高斯初始化、Xavier 初始化和 MSRA 初始化。它们一般都把偏置初始化为 0, 但对权值进行随机初始化。其中, 高斯初始化比较容易理解, 就是根据某个高斯分布来初始化权值, 但均值通常选 0, 方差需要按经验人工选择。下面对 Xavier 和 MSRA 做进一步的说明。

Xavier 的基本思想是保持信息在神经网络中流动过程的方差不变^[101]。这种方法在实际应用时根据一个均匀分布来初始化权值。如果某个神经元 y_j 有 n 个输入 x_1, x_2, \dots, x_n , 相应的连接权值为 w_{ij} , 则满足关系:

$$y_j = w_{1j}x_1 + w_{2j}x_2 + \dots + w_{nj}x_n + b \quad (2.97)$$

那么采用 Xavier 进行初始化的方法就是:

$$w_{ij} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right], b = 0 \quad (2.98)$$

MSRA 的基本思想与 Xavier 类似^[102], 但主要是针对 ReLU 和 PReLU 激活函数来设计的。这种方法在实际应用时根据一个方差为 $\sigma = \sqrt{2/n}$ 的零均匀高斯分布来初始化权值, 即

$$w_{ij} \sim N\left(0, \frac{2}{n}\right), b = 0 \quad (2.99)$$

2.14 丢失输出

在训练神经网络时, 如果训练样本较少, 一般就需要考虑采用某些正则化技巧来防止过拟合。丢失输出 (dropout) 是一种简单有效的正则化技巧, 其基本思想是通过阻止特征检测器的共同作用来提高神经网络的泛化能力^[40]。

丢失输出是指在神经网络的训练过程中随机让网络的某些节点 (包括输入和隐含节点) 不工作。那些不工作的节点可以暂时认为不是网络结构的一部分。例如, 图 2.2 所示的神经网络, 在经过丢失输出处理后, 可能暂时变成图 2.9 所示的结构。

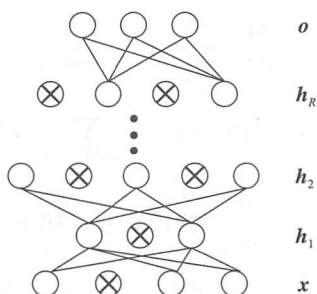


图 2.9 丢失输出神经网络举例，其中带有标记“×”的神经元输出被暂时丢失了

如果用 $y = f(wz + b)$ 表示某层的神经元输出，那么对该层进行丢失输出操作实际上相当于把 y 逐元乘以一个同样大小的随机掩膜 r ，因此输出被修正为

$$\hat{y} = r \circ y = r \circ f(wz + b) \quad (2.100)$$

其中， $r = (r_i)$ ， r_i 服从伯努力分布，即 $r_i \sim \text{Bernoulli}(p)$ 。

丢失输出在本质上可以看作给网络增加了一个具有随机性的辅助层，其计算过程可参见图 2.10a。

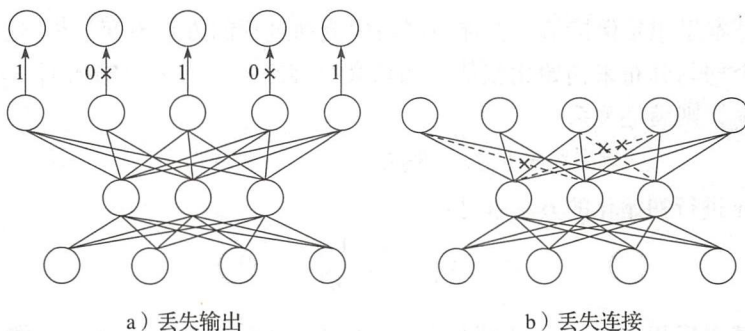


图 2.10 丢失输出和丢失连接的计算过程比较

2.15 丢失连接

丢失连接 (dropconnect) 是对丢失输出的一种简单改进^[41]，两者的区别在于前者用随机掩膜来限制某个层的输出，而后者用其来限制某个层的连接权值。丢失连接在本质上是指在神经网络的训练过程中随机让网络的某些连接不工作，其计算过程如图 2.9b 所示。

如果用 $y = f(wz + b)$ 表示某层的神经元输出，那么对该层进行丢失连接操作实际上相当于把权值矩阵 w 逐元乘以一个同样大小的随机掩膜 r ，因此输出被修正为

$$\hat{y} = f((r \circ w)z + b) \quad (2.101)$$

其中， $r = (r_{ij})$ ， r_{ij} 服从伯努力分布，即 $r_{ij} \sim \text{Bernoulli}(p)$ 。

2.16 随机梯度下降算法

采用严格的反向传播算法训练神经网络，需要同时考虑所有样本对梯度的贡献。如果样本的数量很大，那么梯度下降的每一次迭代都可能花费很长时间，从而可能导致整个过程收敛得非常缓慢。

随机梯度下降（Stochastic Gradient Descent, SGD），或称为增量梯度下降（incremental gradient descent），是一种对梯度下降优化方法的随机近似。其应用条件是目标函数能够表示成一组可微函数之和。而神经网络刚好满足这个条件，所以可以应用随机梯度下降。对神经网络来说，随机梯度下降有两种基本模式：在线和迷你块（mini-batch）。在线模式是先把所有样本随机洗牌，再逐一计算每个样本对梯度的贡献去更新权值，即

$$\mathbf{w} = \mathbf{w} - \eta \frac{\partial e_l}{\partial \mathbf{w}}, l = 1, 2, \dots, L \quad (2.102)$$

其中， e_l 表示网络计算样本 \mathbf{x}_l 的实际输出与期望输出之间的误差。注意，这个按样本逐一更新权值的过程可能需要循环多次。

在线模式的缺点是梯度下降的过程不太稳定、波动较大。一种折中的方法是采用“迷你块”模式，实际上就是把所有样本随机洗牌后分为若干大小为 m 的块，再逐一计算每块对梯度的贡献去更新权值，即

$$\mathbf{w} = \mathbf{w} - \eta \left[\frac{1}{m} \sum_{l=(i-1) \cdot m+1}^{i \cdot m} \frac{\partial e_l}{\partial \mathbf{w}} \right], i = 1, 2, \dots, \lfloor L/m \rfloor \quad (2.103)$$

为了改善随机梯度下降的训练效果，还常常使用权值衰减（weight decay）系数 λ ，可得到

$$\mathbf{w} = (1 - \lambda) \mathbf{w} - \eta \left[\frac{1}{m} \sum_{l=(i-1) \cdot m+1}^{i \cdot m} \frac{\partial e_l}{\partial \mathbf{w}} \right], i = 1, 2, \dots, \lfloor L/m \rfloor \quad (2.104)$$

为了进一步提高稳定性，可以再引入一个动量项 \mathbf{d} 及其加权系数 ν ，得到 SGD 的基本动量模式：

$$\begin{cases} \mathbf{d}_{t+1} = \nu \mathbf{d}_t - \eta \left[\frac{1}{m} \sum_{l=(i-1) \cdot m+1}^{i \cdot m} \frac{\partial e_l}{\partial \mathbf{w}_t} \right], i = 1, 2, \dots, \lfloor L/m \rfloor, \nu > 0 \\ \mathbf{w}_{t+1} = (1 - \lambda) \mathbf{w}_t + \mathbf{d}_{t+1} = (1 - \lambda) \mathbf{w}_t + \nu \mathbf{d}_t - \eta \left[\frac{1}{m} \sum_{l=(i-1) \cdot m+1}^{i \cdot m} \frac{\partial e_l}{\partial \mathbf{w}_t} \right] \end{cases} \quad (2.105)$$

随机梯度下降还有很多其他变种，主要包括 Nesterov 动量模式^[103]、Adagrad^[104]、Adadelta^[105]、RMSProp^[106] 和 Adam^[107] 等。Adam 是目前最好的算法，在不知道如何选择时就选它。

2.17 块归一化

块归一化（batch normalization），又称为批量归一化。对神经网络的训练过程进行块归一化，不仅可以提高网络的训练速度，还可以提高网络的泛化能力^[67]。块归一化可以理解为把对输入数据的归一化扩展到对其他层的输入数据进行归一化，以减小内部数据分布偏移（internal covariate shift）的影响。经过块归一化后，一方面可以通过选择比较大的初始学习

率极大提升训练速度，另一方面还可以不用太关心初始化方法和正则化技巧的选择，从而减少对网络训练过程的人工干预。

块归一化在理论上可以作用于任何变量，但在神经网络中一般直接作用隐含单元的输入。不妨设 x 表示某个隐含单元的输入。 $B = \{x_1, x_2, \dots, x_m\}$ 表示 x 的一个取值块。块归一化实际上是学习一个包含两个参数 γ 和 β 的变换 $BN_{\gamma, \beta}$ ，把 x_i 变成 y_i ，计算过程如下：

$$\begin{cases} \mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \\ \sigma_B \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \\ \hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \\ y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i) \end{cases} \quad (2.106)$$

在学习训练完成之后，令均值 $E[x] = E_B[\mu_B]$ 、方差 $V[x] = \frac{m}{m-1} E_B[\sigma_B^2]$ ，并把变换 $y = BN_{\gamma, \beta}(x)$ 替换成下面的推理尺度变换：

$$y = \frac{\gamma}{\sqrt{V[x] + \varepsilon}} \cdot x + \left(\beta - \frac{\gamma E[x]}{\sqrt{V[x] + \varepsilon}} \right) \quad (2.107)$$

在应用时，块归一化通常被直接作用于某个或多个隐含层的所有输入。

2.18 动态规划算法

动态规划 (dynamic programming) 是运筹学的一个分支，是求解多阶段决策过程 (decision process) 的最优化数学方法^[108]，其核心是贝尔曼最优化原理和贝尔曼方程。

如果用 x_t 表示在 t 时刻的状态，用 $a_t \in \Gamma(x_t)$ 表示 t 时刻的行为，用 $F(x_t, a_t)$ 表示 t 时刻的收益，用 $x_{t+1} = T(x_t, a_t)$ 表示下一时刻的新状态，用 β 表示折扣因子，那么一个从 $t=0$ 开始的无限时域决策问题可以描述最优估值函数 $V(x_0)$ 的数学模型：

$$V(x_0) = \max_{\{a_t\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t F(x_t, a_t) \quad (2.108)$$

求解这个模型的关键在于采用贝尔曼最优化原理：如果 $\{a_t\}_{t=0}^{\infty}$ 是从 $t=0$ 开始的全局最优行为策略，那么 $\forall k > 0$ ， $\{a_t\}_{t=k}^{\infty}$ 一定是从 $t=k$ 开始的全局最优行为策略。

根据这个最优化原理，不难得到下面的贝尔曼方程：

$$V(x_0) = \max_{a_0} \left\{ F(x_0, a_0) + \beta \max_{\{a_t\}_{t=1}^{\infty}} \sum_{t=1}^{\infty} \beta^{t-1} F(x_t, a_t) \right\} = \max_{a_0} \{ F(x_0, a_0) + \beta V(x_1) \} \quad (2.109)$$

其中， $a_0 \in \Gamma(x_0)$ ， $x_1 = T(x_0, a_0)$ 。

卷积神经网络的现代雏形——LeNet

在深度学习的发展史上，卷积神经网络具有举足轻重的地位，对提升计算机视觉的研究水平和实际性能起到了其他方法无可替代的作用。卷积神经网络是受到视觉系统的神经机制启发而提出的模型，其核心思想在于局部感受野和权值共享。1984年日本学者 K. Fukushima 提出的神经认知机被认为是第一个实现的卷积神经网络。1998年，Y. Lecun 等人将卷积层和下采样层相结合，建立了卷积神经网络的现代雏形——LeNet，曾被广泛应用于美国银行支票手写数字识别。本章将先后介绍 LeNet 的原始模型、标准模型、学习算法、Caffe 代码实现及说明、字符识别案例、交通标志识别案例和交通路网提取案例。

3.1 LeNet 的原始模型

第1个真正的卷积神经网络是 Y. Lecun 等人在 1998 年提出的，称为 LeNet^[19]。虽然 LeNet 现在主要指 LeNet5（或 LeNet-5），但也可以包括 LeNet-1 和 LeNet-4 等模型结构。其主要特征是将卷积层和下采样层相结合作为网络的基本结构，共包含 3 个卷积层和 2 个下采样层。最初设计 LeNet 的目的是识别手写字符和打印字符，效果非常好，曾被广泛应用于美国银行支票手写体识别，取得了很大成功。

在 LeNet 的原始模型中，输入是一个矩阵或图像，大小为 32×32 。如果不计输入层，那么这个模型共有 7 层，包括 3 个卷积层、2 个下采样层、1 个全连接层和 1 个输出层，其总体结构如图 3.1 所示。

C1 层是第 1 个卷积层，包含 6 个卷积特征图（convolutional feature map）。每个卷积特征图的大小都是 28×28 ，由一个 5×5 的卷积核（卷积滤波器）对输入图像进行内卷积运算得到，其中每个神经元与输入中相应 5×5 区域相连。C1 层共有 156 个参数，因为每个卷积核有 $5 \times 5 = 25$ 个权值和 1 个偏置，6 个卷积核共有 $(25 + 1) \times 6 = 156$ 个参数，还有 122 304 个连接。

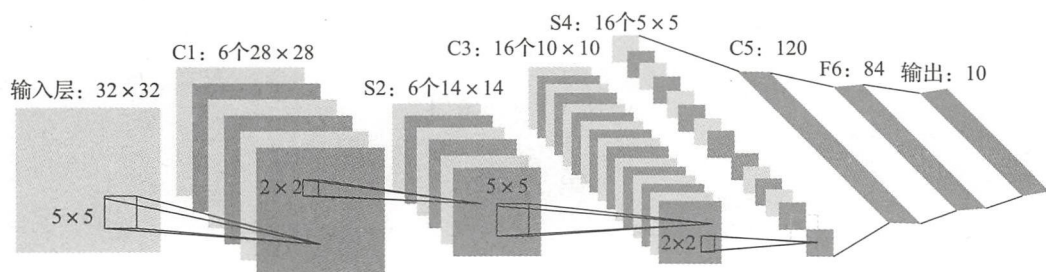


图 3.1 LeNet 的原始模型结构示意图

S2 层是第 1 个下采样层，包含 6 个 14×14 的下采样特征图。每个下采样特征图都是由 C1 层的相应卷积特征图经过大小为 2×2 、步长为 2 的窗口进行平均池化，然后再利用激活函数 sigmoid 进行一次非线性变换处理得到。进行下采样的目的是利用局部相关性减少后续数据处理量，同时又保留有用信息。与 C1 层的卷积特征图相比，S2 层的下采样特征图整体上缩小为原来的 $1/4$ ，行列各缩小为原来的 $1/2$ 。另外，每个下采样特征图还有一个权值参数和一个偏置参数，6 个特征图共包含 12 个可训练参数，以及 5880 个连接。

C3 层是第 2 个卷积层，包含 16 个 10×10 的卷积特征图。在每个卷积特征图中，每个神经元都与 S2 层的若干个 5×5 邻域存在局部连接，这些 5×5 邻域分别位于不同特征图的相同位置上，详细的连接关系如表 3.1 所示。表 3.1 的第 1 行对 C3 的 16 个卷积特征图从 0 到 15 进行编号，第 1 列对 S2 的 6 个下采样特征图从 0 到 5 进行编号。不难看出，在 C3 层的 16 个卷积特征图中，有 6 个的神经元与 S2 层的 3 个下采样特征图存在局部连接（见表 3.1 的第 2 ~ 7 列），有 9 个神经元与 S2 层的 4 个下采样特征图存在局部连接（见表 3.1 的第 8 ~ 15 列），还有 1 个神经元与 S2 层的 6 个下采样特征图都存在局部连接（见表 3.1 的第 16 列）。通过简单的计算易知，C3 层总共有 $6 \times (3 \times 5 \times 5 + 1) + 9 \times (4 \times 5 \times 5 + 1) + 1 \times (6 \times 5 \times 5 + 1) = 1516$ 个参数，还有 151 600 个连接。

表 3.1 在 LeNet 的原始模型中 S2 层到 C3 层的连接关系

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	×				×	×	×			×	×	×	×		×	×
1	×	×				×	×	×			×	×	×	×		×
2	×	×	×				×	×	×			×		×	×	×
3		×	×	×			×	×	×	×			×		×	×
4			×	×	×			×	×	×	×		×	×		×
5				×	×	×			×	×	×	×		×	×	×

S4 层是第 2 个下采样层，包含 16 个 5×5 的下采样特征图。每个下采样特征图都是由 C3 层的相应卷积特征图经过 2×2 不重叠窗口的平均池化，然后再利用激活函数 sigmoid 进行一次非线性变换处理得到。与 C3 层的卷积特征图相比，S4 层的下采样特征图整体上缩小为原来的 $1/4$ ，行列各缩小为原来的 $1/2$ 。另外，每个下采样特征图还有一个权值参数和一个偏置参数，16 个下采样特征图共包含 32 个可训练参数和 2000 个连接。

C5 层是第 3 个卷积层，包含 120 个 1×1 的卷积特征图。其中每个神经元与 S4 层中所有下采样特征图的一个 5×5 邻域相连。由于下采样特征图和卷积核的大小都是 5×5 ，所以 C5 层的卷积特征图大小为 1×1 。这也意味着从 S4 到 C5 实际上是一种全连接。C5 层被标记为卷积层而不是全连接层，是因为 LeNet-5 的输入通过填充补零变大了，但其他部分保持不变，导致 C5 层的特征图大于 1×1 。另外，考虑到每个卷积特征图还有一个偏置，C5 层共有 $(16 \times 5 \times 5 + 1) \times 120 = 48\,120$ 个可训练参数。

F6 层是全连接层，包含 84 个神经元。每个神经元与 C5 层的所有神经元相连，其值由 C5 层的输出向量与 F6 的权重向量做内积，加上一个偏置，再经过尺度化双曲正切函数 (scaled hyperbolic tangent) 的挤压作用得到。F6 层共有 $84 \times 120 + 84 = 10\,164$ 个可训练参数。

最后一层是输出层，由 10 个欧几里得径向函数神经元组成。每个这样的神经元代表一个类别，且都与 F6 层的 84 个神经元相连，其输出值由输入向量和权值向量的欧氏距离平方得到。显然，输出层的可训练参数共有 $84 \times 10 = 840$ 个。

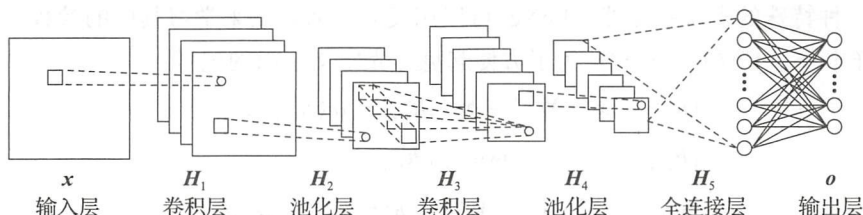


图 3.2 LeNet 的标准模型结构。每个平面表示一个特征图，其中所有神经元共享权值，但偏置可能不同

3.2 LeNet 的标准模型

现在常用的 LeNet 模型与其原始模型已经有所不同，主要区别在于把非线性变换从下采样层移到了卷积层，且把输出层的激活函数从欧几里得径向函数替换成了软最大函数。标准的模型结构如图 3.2 所示。这个标准模型从结构上可以分解为输入层、卷积层、池化层、卷积层、池化层、全连接层和输出层。输入层用 x 表示，通常是一个矩阵或一幅图像。下面是对其他层的详细描述和说明。

$H_1 = (h_{1,\alpha})$ 是第 1 个隐含层，且是一个卷积层 (CL)。其中第 α 个卷积面 $h_{1,\alpha}$ 的卷积核为 $W^{1,\alpha}$ 、偏置为 $b^{1,\alpha}$ ，计算公式为

$$h_{1,\alpha} = \sigma(x \tilde{*} W^{1,\alpha} + b^{1,\alpha}) \quad (3.1)$$

$H_2 = (h_{2,\alpha})$ 是第 2 个隐含层，且是一个池化层。其中第 α 个池化面 $h_{2,\alpha}$ 是通过下采样来计算的，即：

$$h_{2,\alpha} = \text{down}_{\lambda_2, \tau_2}(h_{1,\alpha}) \quad (3.2)$$

$H_3 = (h_{3,\beta})$ 是第 3 个隐含层，也是一个卷积层。其中第 β 个卷积面 $h_{3,\beta}$ 是利用所有池化面 $h_{2,\alpha}$ 和相应的卷积核 $W_{2,\alpha}^{3,\beta}$ 及偏置 $b^{3,\beta}$ 来计算的，即

$$h_{3,\beta} = \sigma\left(\sum_{\alpha} h_{2,\alpha} \tilde{*} W_{2,\alpha}^{3,\beta} + b^{3,\beta}\right) \quad (3.3)$$

$H_4=(h_{4,\beta})$ 是第 4 个隐含层，也是一个池化层。其中第 β 个池化面 $h_{4,\beta}$ 是通过 $h_{3,\beta}$ 的下采样来计算的，即

$$h_{4,\beta} = \text{down}_{\lambda_4, \tau_4}(h_{3,\beta}) \quad (3.4)$$

$H_5=(h_5)$ 是第 5 个隐含层，且是唯一的全连接层。其中 h_5 是利用 $h_{4,\beta}$ 、三维张量 $W^{5,\beta}$ 和偏置 b^5 来计算的，即

$$h_5 = \sigma\left(\sum_{\beta} W^{5,\beta} h_{4,\beta} + b^5\right) \quad (3.5)$$

o 是输出层，其作用是利用 h_5 、权矩阵 W^6 和偏置 b^6 通过软最大函数产生伪概率，计算公式为

$$o = \text{softmax}(W^6 h_5 + b^6) \quad (3.6)$$

3.3 LeNet 的学习算法

作为一种特殊的多层感知器，LeNet 可以用反向传播算法来学习其中的参数。不妨设有 N 个训练样本 $(x^l, y^l) (1 \leq l \leq N)$ 。为了方便起见，先定义下面的记号：

$$\begin{cases} u_{1,\alpha}^l = x^l * W^{1,\alpha} + b^{1,\alpha}, h_{1,\alpha}^l = \sigma(u_{1,\alpha}^l), \\ h_{2,\alpha}^l = u_{2,\alpha}^l = \text{down}_{\lambda_2, \tau_2}(h_{1,\alpha}^l), \\ u_{3,\beta}^l = \sum_{\alpha} h_{2,\alpha}^l * W_{2,\alpha}^{3,\beta} + b^{3,\beta}, h_{3,\beta}^l = \sigma(u_{3,\beta}^l), \\ h_{4,\beta}^l = u_{4,\beta}^l = \text{down}_{\lambda_4, \tau_4}(h_{3,\beta}^l), \\ u_5^l = \sum_{\beta} W^{5,\beta} h_{4,\beta}^l + b^5, h_5^l = \sigma(u_5^l), \\ u_6^l = W^6 h_5^l + b^6, o^l = \text{softmax}(u_6^l) \end{cases} \quad (3.7)$$

如果用 $\text{diag}(o^l)$ 表示以 o^l 各分量为对角元的对角矩阵，且选择平方误差 (2.57) 为目标函数，那么通过链式法则可以计算得到各层反向传播误差，如下：

$$\begin{cases} \delta_6^l = \frac{\partial E}{\partial u_6^l} = \frac{\partial E}{\partial o^l} \cdot \frac{\partial o^l}{\partial u_6^l} = [\text{diag}(o^l) - o^l (o^l)^T] (o^l - y^l), \\ \delta_5^l = \frac{\partial E}{\partial u_5^l} = \frac{\partial E}{\partial u_6^l} \cdot \frac{\partial u_6^l}{\partial u_5^l} = [(W^6)^T \delta_6^l] \circ \sigma'(u_5^l), \\ \delta_{4,\beta}^l = \frac{\partial E}{\partial u_{4,\beta}^l} = \frac{\partial E}{\partial u_5^l} \cdot \frac{\partial u_5^l}{\partial u_{4,\beta}^l} = (W^{5,\beta})^T \delta_5^l, \\ \delta_{3,\beta}^l = \frac{\partial E}{\partial u_{3,\beta}^l} = \frac{\partial E}{\partial u_{4,\beta}^l} \cdot \frac{\partial u_{4,\beta}^l}{\partial u_{3,\beta}^l} = \left[\frac{1}{\lambda_4 \times \tau_4} \text{up}_{\lambda_4 \times \tau_4}(\delta_{4,\beta}^l) \right] \circ \sigma'(u_{3,\beta}^l), \\ \delta_{2,\alpha}^l = \frac{\partial E}{\partial u_{2,\alpha}^l} = \frac{\partial E}{\partial u_{3,\beta}^l} \cdot \frac{\partial u_{3,\beta}^l}{\partial u_{2,\alpha}^l} = \delta_{3,\beta}^l * \text{rot180}(W_{2,\alpha}^{3,\beta}), \\ \delta_{1,\alpha}^l = [\text{up}_{\lambda_2 \times \tau_2}(\delta_{2,\alpha}^l) / (\lambda_2 \times \tau_2)] \circ \sigma'(u_{1,\alpha}^l). \end{cases} \quad (3.8)$$

而且, 不难进一步得到梯度:

$$\begin{cases} \frac{\partial E}{\partial \mathbf{W}^6} = \sum_{l=1}^N \delta_6^l (\mathbf{h}_5^l)^\top, \frac{\partial E}{\partial \mathbf{b}^6} = \sum_{l=1}^N \delta_6^l, \\ \frac{\partial E}{\partial \mathbf{W}^{5,\beta}} = \sum_{l=1}^N \delta_5^l (\mathbf{h}_{4,\beta}^l)^\top, \frac{\partial E}{\partial \mathbf{b}^5} = \sum_{l=1}^N \delta_5^l, \\ \frac{\partial E}{\partial \mathbf{W}_{2,\alpha}^{3,\beta}} = \sum_{l=1}^N \mathbf{h}_{2,\alpha}^l * \delta_{3,\beta}^l, \frac{\partial E}{\partial \mathbf{b}^{3,\beta}} = \sum_{l=1}^N \delta_{3,\beta}^l, \\ \frac{\partial E}{\partial \mathbf{W}^{1,\alpha}} = \sum_{l=1}^N \mathbf{x}^l * \delta_{1,\alpha}^l, \frac{\partial E}{\partial \mathbf{b}^{1,\alpha}} = \sum_{l=1}^N \delta_{1,\alpha}^l \end{cases} \quad (3.9)$$

基于以上推导和计算, 很容易利用梯度下降的思想给图 3.2 的 LeNet 建立反向传播算法, 即算法 3.1。

算法 3.1 LeNet 的反向传播算法

输入: 训练集 $\{(\mathbf{x}^l, \mathbf{y}^l), 1 \leq l \leq L\}$, 学习率 η , 网络结构, 迭代次数 epoch

输出: 所有权值和偏置

1. 随机初始化 $\mathbf{W}^{1,\alpha} \approx \mathbf{0}$, $\mathbf{b}^{1,\alpha} \approx \mathbf{0}$, $\mathbf{W}_{2,\alpha}^{3,\beta} \approx \mathbf{0}$, $\mathbf{b}^{3,\beta} \approx \mathbf{0}$, $\mathbf{W}^{5,\beta} \approx \mathbf{0}$, $\mathbf{b}^5 \approx \mathbf{0}$, $\mathbf{W}^6 \approx \mathbf{0}$, $\mathbf{b}^6 \approx \mathbf{0}$;
2. **for** $i = 1, 2, \dots, \text{epoch}$ **do**
3. **for** $l = 1, 2, \dots, L$ **do**
4. 用(3.7)和(3.8)计算 $\delta_{1,\alpha}^l, \delta_{2,\alpha}^l, \delta_{3,\beta}^l, \delta_{4,\beta}^l, \delta_5^l, \delta_6^l$;
5. **end for**
6. 用(3.9)计算 $\frac{\partial E}{\partial \mathbf{W}^6}, \frac{\partial E}{\partial \mathbf{b}^6}, \frac{\partial E}{\partial \mathbf{W}^{5,\beta}}, \frac{\partial E}{\partial \mathbf{b}^5}, \frac{\partial E}{\partial \mathbf{W}_{2,\alpha}^{3,\beta}}, \frac{\partial E}{\partial \mathbf{b}^{3,\beta}}, \frac{\partial E}{\partial \mathbf{W}^{1,\alpha}}, \frac{\partial E}{\partial \mathbf{b}^{1,\alpha}}$;
7. 如果梯度的模足够小, 则停止;
8. 否则, 用学习率和梯度更新权值和偏置:
9. $\mathbf{W}^6 = \mathbf{W}^6 - \eta \frac{\partial E}{\partial \mathbf{W}^6}$, $\mathbf{b}^6 = \mathbf{b}^6 - \eta \frac{\partial E}{\partial \mathbf{b}^6}$, $\mathbf{W}^{5,\beta} = \mathbf{W}^{5,\beta} - \eta \frac{\partial E}{\partial \mathbf{W}^{5,\beta}}$, $\mathbf{b}^5 = \mathbf{b}^5 - \eta \frac{\partial E}{\partial \mathbf{b}^5}$,
10. $\mathbf{W}_{2,\alpha}^{3,\beta} = \mathbf{W}_{2,\alpha}^{3,\beta} - \eta \frac{\partial E}{\partial \mathbf{W}_{2,\alpha}^{3,\beta}}$, $\mathbf{b}^{3,\beta} = \mathbf{b}^{3,\beta} - \eta \frac{\partial E}{\partial \mathbf{b}^{3,\beta}}$, $\mathbf{W}^{1,\alpha} = \mathbf{W}^{1,\alpha} - \eta \frac{\partial E}{\partial \mathbf{W}^{1,\alpha}}$, $\mathbf{b}^{1,\alpha} = \mathbf{b}^{1,\alpha} - \eta \frac{\partial E}{\partial \mathbf{b}^{1,\alpha}}$;
11. **end for**

如果把目标函数改为交叉熵 (2.61), 那么相应的反向传播算法只需把 δ_6^l 的计算公式改为

$$\delta_6^l = \frac{\partial E}{\partial \mathbf{u}_6^l} = \frac{\partial E}{\partial \mathbf{o}^l} \cdot \frac{\partial \mathbf{o}^l}{\partial \mathbf{u}_6^l} = (\text{diag}(\mathbf{1} ./ (\mathbf{1} - \mathbf{o}^l)) - (\mathbf{o}^l)^\top (\mathbf{1} ./ (\mathbf{1} - \mathbf{o}^l))) (\mathbf{o}^l - \mathbf{y}^l) \quad (3.10)$$

其他部分保持不变。注意, 在 (3.10) 式中, $\mathbf{1} ./ (\mathbf{1} - \mathbf{o}^l)$ 表示列向量 $\mathbf{1}$ 与 $(\mathbf{1} - \mathbf{o}^l)$ 对应元素相除得到的列向量。

最后，还必须指出，在输出为软最大函数时，卷积神经网络一般都使用近似交叉熵 (2.66) 作为目标函数，以加快计算速度。如果期望输出 \mathbf{y}^l 仅有一个分量为 $y_{ji}^l = 1$ ，那么相应的 δ_o^l 可以计算如下：

$$\delta_o^l = o^l - y^l \quad (3.11)$$

3.4 LeNet 的 Caffe 代码实现及说明

LeNet 是 Caffe 安装包自带的一个例子模型。在安装好 Caffe 包后，其 Caffe 代码可以在子文件夹 examples 下的 mnist 案例中找到。

LeNet 的 Caffe 实现代码共有三个文件：网络结构文件 lenet_train_test.prototxt、求解器配置文件 lenet_solver.prototxt 和伪概率计算文件 lenet.prototxt。其中，lenet_train_test.prototxt 用来定义网络的训练数据目录、测试数据目录和网络结构细节，包括每个卷积层的卷积核个数、大小、步长，每个下采样层的类型、窗口大小、步长，全连接层的节点数目、激活函数的类型、损失函数类型，以及层与层之间的关系等。lenet_solver.prototxt 用来给求解器配置训练和测试网络的有关超参数，包括学习率的大小、训练的迭代次数及优化方法、使用的计算模式（CPU 或 GPU）等。lenet.prototxt 用来在模型训练好后对未知样本计算分类伪概率。下面分别对这三个文件的具体内容进行描述。

1. 网络结构文件 lenet_train_test.prototxt 的内容描述

该文件主要用来定义 LeNet 的网络结构，有关参数及其含义见表 3.2。

表 3.2 网络结构文件 lenet_train_test.prototxt 的参数及其含义

参数	含义
CaseName	案例名称
TrainSetDir	训练集所在目录
Train_batch_size	训练集迷你块的大小
TestSetDir	测试集所在目录
Test_batch_size	测试集迷你块的大小
conv1_num_output	卷积面的个数
conv1_kernel_size	卷积核的大小
conv1_stride	卷积核的移动步长
pool1_kernel_size	池化窗的大小
pool1_stride	池化窗的移动步长
conv2_num_output	卷积面的个数
conv2_kernel_size	卷积核的大小
conv2_stride	卷积核的移动步长
pool2_kernel_size	池化窗的大小
pool2_stride	池化窗的移动步长
ip1_num_output	第 1 个全连接层的节点数
ip2_num_output	第 2 个全连接层的节点数

下面是具体的代码。

```
name: "LeNet"
layer {
  name: "CaseName"                # 表示案例名称
  type: "Data"                    # 表示该层用于数据输入
  top: "data"
  top: "label"
  data_param {
    source: "TrainSetDir"          # 表示训练集目录
    backend: LEVELDB              # 一种 Caffe 数据类型
    batch_size: Train_batch_size  # 表示训练集迷你块的大小
  }
  transform_param {
    scale: 0.00390625             # 将图像灰度值除以 256 归一化, 把输入数据变换到 [0,1] 区间
  }
  include: { phase: TRAIN }
}
layer {
  name: "CaseName"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "TestSetDir"          # 表示测试集目录
    backend: LEVELDB
    batch_size: Test_batch_size  # 表示测试集迷你块的大小
  }
  transform_param {
    scale: 0.00390625
  }
  include: { phase: TEST }
}
layer {
  name: "conv1"
  type: "Convolution"            # 表示该层为卷积层
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1                   # 权重学习率倍数, 用来放大初始学习率 base_lr
  }
  param {
    lr_mult: 2                   # 偏置学习率倍数, 把初始学习率 base_lr 放大 2 倍
  }
  convolution_param {
    num_output: conv1_num_output # 表示卷积面的个数
    kernel_size: conv1_kernel_size # 表示卷积核的大小
    stride: conv1_stride         # 表示卷积核的移动步长
    weight_filler {
      type: "xavier" # 使用 xavier 方法初始化权值, 也可以设置为 "gaussian" 或其他方法
    }
    bias_filler {
```

```

        type: "constant" # 表示把偏置初始化为常数 0, 若要初始化为 0.1, 可用语句 value: 0.1
    }
}
}
layer {
    name: "relu1"
    type: "Insanity"      # 把该层的激活函数设置为 Insanity 类型, 即一种 randomized LReLU
    bottom: "conv1"
    top: "conv1"
}
layer {
    name: "pool1"
    type: "Pooling"       # 池化层
    bottom: "conv1"
    top: "pool1"
    pooling_param {
        pool: MAX          # 把池化类型选为最大池化, 还可选择平均池化
        kernel_size: pool1_kernel_size # 表示池化窗的大小
        stride: pool1_stride # 表示池化窗的移动步长
    }
}
layer {
    name: "conv2"
    type: "Convolution"   # 卷积层
    bottom: "pool1"
    top: "conv2"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    convolution_param {
        num_output: conv2_num_output # 表示卷积面的个数
        kernel_size: conv2_kernel_size # 表示卷积核的大小
        stride: conv2_stride          # 表示卷积核的移动步长
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
layer {
    name: "relu2"
    type: "Insanity"
    bottom: "conv2"
    top: "conv2"
}
layer {
    name: "pool2"

```



```

type: "Pooling"                                # 池化层
bottom: "conv2"
top: "pool2"
pooling_param {
    pool: MAX
    kernel_size: pool2_kernel_size # 表示池化窗的大小
    stride: pool2_stride           # 表示池化窗的移动步长
}
}
layer {
    name: "ip1"
    type: "InnerProduct"              # 全连接层, 在这里也称为内积
    bottom: "pool2"
    top: "ip1"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    inner_product_param {
        num_output: ip1_num_output    # 表示全连接层的节点数
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
}
layer {
    name: "relu3"
    type: "Insanity"
    bottom: "ip1"
    top: "ip1"
}
layer {
    name: "ip2"
    type: "InnerProduct"              # 全连层
    bottom: "ip1"
    top: "ip2"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    inner_product_param {
        num_output: ip2_num_output    # 全连接层的节点数
        weight_filler {
            type: "xavier"
        }
    }
}

```

```

        bias_filler {
            type: "constant"
        }
    }
}
layer {
    name: "accuracy"
    type: "Accuracy"
    bottom: "ip2"
    bottom: "label"
    top: "accuracy"
    include {
        phase: TEST
    }
}
layer {
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "ip2"
    bottom: "label"
    top: "loss"
}

```

该层只出现在测试阶段，用于计算准确率
 # 计算准确率要用到全连接层 ip2
 # 计算准确率还要用到标签层

带损失的软最大输出层，采用退化交叉熵损失函数
 # 计算损失要用到全连接层 ip2
 # 计算损失还要用到标签层

2. 求解器配置文件 lenet_solver.prototxt 的内容描述

该文件主要用于配置对 LeNet 模型进行训练和测试的有关参数，其含义在表 3.3 中给出了详细描述。文件对这些参数的一种具体配置情况如方框 3.1 所示。

表 3.3 求解器配置文件 lenet_solver.prototxt 的超参数及其含义

超参数	含义
net	网络结构文件的位置
test_iter	测试次数（测试样本数 /Batch_size）
test_interval	测试间隔（每测试一次经过的训练次数）
base_lr	初始学习率
momentum	动量项的加权系数
weight_decay	权值衰减系数
lr_policy	学习率的下降策略
gamma	学习率的策略参数
Power	学习率的策略参数
display	显示间隔（每显示一次中间结果经过的训练次数）
max_iter	最大训练次数
snapshot	保存间隔（每保存一次结果经过的训练次数）
snapshot_prefix	保存训练结果的目录
type	优化算法的类型
solver_mode	处理模式（GPU 或 CPU）

方框 3.1 求解器的一种具体超参数配置

```

net: "C:/Caffe/caffe-windows-master/examples/mnist/lenet_train_test.prototxt"
test_iter: 100
test_interval: 500 # 每训练 500 次，测试 1 次
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
lr_policy: "inv" # 把学习率的下降策略选为 inv 类型，还可以选为 fixed、step 和 exp 等类型
gamma: 0.0001
power: 0.75
display: 100 # 每训练 100 次显示 1 次中间结果
max_iter: 10000 # 最多训练 10 000 次
snapshot: 5000 # 每训练 5000 次保存 1 次结果，命名为 _iter_5000.caffemodel 等
snapshot_prefix: "C:/Caffe/caffe-windows-master/examples/mnist/lenet"
type: "SGD" # 选用随机梯度下降法训练，还可选 AdaDelta、AdaGrad、Adam、
# Nesterov、RMSProp
solver_mode: GPU

```

3. 伪概率计算文件 lenet.prototxt 的内容描述

该文件主要用来在模型训练好后对未知样本计算分类伪概率，有关参数及其含义见表 3.4。

表 3.4 伪概率计算文件 lenet.prototxt 的参数及含义

参数	含义
dim1、dim2、dim3、dim4	输入层大小
conv1_num_output	卷积面的个数
conv1_kernel_size	卷积核的大小
conv1_stride	卷积核的移动步长
pool1_kernel_size	池化窗的大小
pool1_stride	池化窗的移动步长
conv2_num_output	卷积面的个数
conv2_kernel_size	卷积核的大小
conv2_stride	卷积核的移动步长
pool2_kernel_size	池化窗的大小
pool2_stride	池化窗的移动步长
ip1_num_output	第 1 个全连接层的节点数
ip2_num_output	第 2 个全连接层的节点数

下面是具体的代码。

```

name: "LeNet"
layer {
  name: "data"
  type: "Input"
  top: "data"

```



```

input_param { shape: { dim: dim1 dim: dim2 dim: dim3 dim: dim4 } }
# 表示输入层大小
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: conv1_num_output # 表示卷积面的个数
    kernel_size: conv1_kernel_size # 表示卷积核的大小
    stride: conv1_stride # 表示卷积核的移动步长
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: pool1_kernel_size # 表示池化窗的大小
    stride: pool1_stride # 表示池化窗的移动步长
  }
}
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: conv2_num_output
    kernel_size: conv2_kernel_size

```

```

        stride: conv2_stride
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
layer {
    name: "pool2"
    type: "Pooling"
    bottom: "conv2"
    top: "pool2"
    pooling_param {
        pool: MAX
        kernel_size: pool2_kernel_size
        stride: pool2_stride
    }
}
layer {
    name: "ip1"
    type: "InnerProduct"
    bottom: "pool2"
    top: "ip1"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    inner_product_param {
        num_output: ip1_num_output # 表示全连层的节点数
    }
    weight_filler {
        type: "xavier"
    }
    bias_filler {
        type: "constant"
    }
}
layer {
    name: "relu1"
    type: "ReLU"
    bottom: "ip1"
    top: "ip1"
}
layer {
    name: "ip2"
    type: "InnerProduct"
    bottom: "ip1"

```

```

top: "ip2"
param {
    lr_mult: 1
}
param {
    lr_mult: 2
}
inner_product_param {
    num_output: ip2_num_output      # ip2_num_output 表示全连层的节点数
    weight_filler {
        type: "xavier"
    }
    bias_filler {
        type: "constant"
    }
}
}
layer {
    name: "prob"
    type: "Softmax"
    bottom: "ip2"
    top: "prob"
}
}

```

最后，需要指出，这个伪概率计算文件 `lenet.prototxt` 是 Caffe 安装包自带的，但所定义的网络结构与网络结构文件 `lenet_train_test.prototxt` 并不完全一致。因此，如果模型是基于 `lenet_train_test.prototxt` 训练的，还必须对 `lenet.prototxt` 文件进行适当修改，使得两者的模型结构一致，才能对未知样本正确计算伪概率。

3.5 LeNet 的手写数字识别案例

本节将描述一个利用 LeNet 对手写数字进行识别的案例，其中用到的 MNIST 数据集可以根据表 1.2 提供的地址下载。这个数据集需要利用方框 3.2 的 `create_mnist.bat` 文件转化为特殊的 Caffe 格式 LEVELDB，然后分别存放到训练文件夹 `mnist-train-leveldb` 和测试文件夹 `mnist-test-leveldb`。为了利用 LeNet 进行手写数字识别，还需要在 `lenet_train_test.prototxt` 和 `lenet_solver.prototxt` 文件中根据表 3.5 和表 3.6 设置各个参数的值。

方框 3.2 文件 `create_mnist.bat` 的内容

```

rd /s /q mnist_train_leveldb
rd /s /q mnist_test_leveldb
convert_mnist_data.exe data/train-images.idx3-ubyte data/train-labels.idx1-
ubyte mnist_train_leveldb --backend=leveldb
convert_mnist_data.exe data/t10k-images.idx3-ubyte data/t10k-labels.idx1-ubyte
mnist_test_leveldb --backend=leveldb
Pause

```


表 3.5 手写数字识别案例在 lenet_train_test.prototxt 文件中设置的参数值

参数	设置值
CaseName	MNIST
TrainSetDir	C:/Caffe/caffe-windows-master/examples/mnist/mnist-train-leveldb
Train_batch_size	64
TestSetDir	C:/Caffe/caffe-windows-master/examples/mnist/mnist-test-leveldb
Train_batch_size	100
conv1_num_output	20
conv1_kernel_size	5
conv1_stride	1
pool1_kernel_size	2
pool1_stride	2
conv2_num_output	50
conv2_kernel_size	5
conv2_stride	1
pool2_kernel_size	2
pool2_stride	2
ip1_num_output	500
ip2_num_output	10

表 3.6 手写数字识别案例在 lenet_solver.prototxt 文件中设置的超参数值

超参数	设置值
lenet_train_test.prototxt_dir	C:/Caffe/caffe-windows-master/examples/mnist/lenet_train_test.prototxt
test_iter	100
test_interval	500
base_lr	0.01
momentum	0.9
weight_decay	0.000 5
lr_policy	inv
gamma	0.000 1
power	0.75
display	100
max_iter	10 000
snapshot	5 000
snapshot_prefix	C:/Caffe/caffe-windows-master/examples/mnist/lenet
solver_mode	CPU



设置好参数后，参照图 3.3，在命令行窗口中输入有关命令，按回车键，程序开始运行。程序运行后，将依次出现一系列信息窗口界面，如图 3.4～图 3.7 所示。

图 3.3 手写数字识别案例的运行命令

图 3.4 手写数字识别案例运行后显示的网络结构信息

图 3.4 显示的网络结构信息其实就是对文件 `lenet_train_test.prototxt` 的加载过程，而图 3.5 显示的则是网络各层的详细创建过程。在创建完成后，就开始根据文件 `lenet_solver.prototxt` 中设置的参数对网络权值和偏置进行训练，由于 `solver_mode` 选为 CPU，所以训练过程仅采用 CPU，不使用 GPU。图 3.6 显示的是在训练网络的过程中，随着迭代次数的增加，学习率、训练损失和测试损失、训练准确率和测试准确率的变化。在本案例中，每迭代 100 次显示一次训练情况，每迭代 500 次显示一次测试情况，总共训练 10 000 次。训练完成时的最终结果如图 3.7 所示，训练损失函数值为 0.007 162 71，测试损失函数值为 0.026 721 2，测试准确率为 99.17%。




```

C:\Windows\system32\cmd.exe
10510 20:19:09.727176 15580 layer_factory.hpp:77] Creating layer conv2
10510 20:19:09.732139 15580 net.cpp:91] Creating Layer conv2
10510 20:19:09.733192 15580 net.cpp:425] conv2 <- pool1
10510 20:19:09.733192 15580 net.cpp:399] conv2 -> conv2
10510 20:19:09.734194 15580 net.cpp:141] Setting up conv2
10510 20:19:09.734194 15580 net.cpp:148] Top shape: 64 50 8 8 (204800)
10510 20:19:09.735198 15580 net.cpp:156] Memory required for data: 5443340
10510 20:19:09.735198 15580 layer_factory.hpp:77] Creating layer pool2
10510 20:19:09.736199 15580 net.cpp:91] Creating Layer pool2
10510 20:19:09.736199 15580 net.cpp:425] pool2 <- conv2
10510 20:19:09.737202 15580 net.cpp:399] pool2 -> pool2
10510 20:19:09.737202 15580 net.cpp:141] Setting up pool2
10510 20:19:09.738205 15580 net.cpp:148] Top shape: 64 50 4 4 (51200)
10510 20:19:09.738205 15580 net.cpp:156] Memory required for data: 5643640
10510 20:19:09.740211 15580 layer_factory.hpp:77] Creating layer relu2
10510 20:19:09.742216 15580 net.cpp:91] Creating Layer relu2
10510 20:19:09.743218 15580 net.cpp:425] relu2 <- pool2
10510 20:19:09.743218 15580 net.cpp:386] relu2 -> pool2 (in-place)
10510 20:19:09.744221 15580 net.cpp:141] Setting up relu2
10510 20:19:09.745224 15580 net.cpp:148] Top shape: 64 50 4 4 (51200)
10510 20:19:09.745224 15580 net.cpp:156] Memory required for data: 5853440
10510 20:19:09.746227 15580 layer_factory.hpp:77] Creating layer ip1
10510 20:19:09.747229 15580 net.cpp:91] Creating Layer ip1
10510 20:19:09.747229 15580 net.cpp:425] ip1 <- pool2
10510 20:19:09.748239 15580 net.cpp:399] ip1 -> ip1
10510 20:19:09.752243 15580 net.cpp:141] Setting up ip1
10510 20:19:09.752243 15580 net.cpp:148] Top shape: 64 500 (32000)
10510 20:19:09.754248 15580 net.cpp:156] Memory required for data: 5981440
10510 20:19:09.755252 15580 layer_factory.hpp:77] Creating layer relu3
搜狗拼音输入法 全 :53 15580 net.cpp:91] Creating Layer relu3

```

图 3.5 手写数字识别案例运行后显示的网络结构各层详细信息

```

C:\Windows\system32\cmd.exe
10510 20:24:56.273475 15580 solver.cpp:228] Iteration 6200, loss = 0.0375226
10510 20:24:56.274478 15580 solver.cpp:244] Train net output #0: loss = 0.0375229 (* 1 = 0.0375229 loss)
10510 20:24:56.277487 15580 sgd_solver.cpp:106] Iteration 6200, lr = 0.00696408
10510 20:25:01.059207 15580 solver.cpp:228] Iteration 6300, loss = 0.00789029
10510 20:25:01.060210 15580 solver.cpp:244] Train net output #0: loss = 0.00789055 (* 1 = 0.00789055 loss)
10510 20:25:01.061213 15580 sgd_solver.cpp:106] Iteration 6300, lr = 0.00693201
10510 20:25:06.046476 15580 solver.cpp:228] Iteration 6400, loss = 0.0135513
10510 20:25:06.047473 15580 solver.cpp:244] Train net output #0: loss = 0.0135521 (* 1 = 0.0135521 loss)
10510 20:25:06.049484 15580 sgd_solver.cpp:106] Iteration 6400, lr = 0.00690029
10510 20:25:10.822180 15580 solver.cpp:327] Iteration 6500, Testing net (#0)
10510 20:25:13.883349 15580 solver.cpp:404] Test net output #0: accuracy = 0.9904
10510 20:25:13.884351 15580 solver.cpp:404] Test net output #1: loss = 0.0284784 (* 1 = 0.0284784 loss)
10510 20:25:13.941479 15580 solver.cpp:228] Iteration 6500, loss = 0.0123931
10510 20:25:13.942482 15580 solver.cpp:244] Train net output #0: loss = 0.0123934 (* 1 = 0.0123934 loss)
10510 20:25:13.943485 15580 sgd_solver.cpp:106] Iteration 6500, lr = 0.0068689
10510 20:25:18.910698 15580 solver.cpp:228] Iteration 6600, loss = 0.0378703
10510 20:25:18.910698 15580 solver.cpp:244] Train net output #0: loss = 0.0378706 (* 1 = 0.0378706 loss)
10510 20:25:18.911702 15580 sgd_solver.cpp:106] Iteration 6600, lr = 0.00683784
10510 20:25:23.847833 15580 solver.cpp:228] Iteration 6700, loss = 0.0160045
10510 20:25:23.848837 15580 solver.cpp:244] Train net output #0: loss = 0.0160048 (* 1 = 0.0160048 loss)
10510 20:25:23.849840 15580 sgd_solver.cpp:106] Iteration 6700, lr = 0.00680711
10510 20:25:23.773939 15580 solver.cpp:228] Iteration 6800, loss = 0.0030395
10510 20:25:23.774968 15580 solver.cpp:244] Train net output #0: loss = 0.00303979 (* 1 = 0.00303979 loss)
10510 20:25:23.775977 15580 sgd_solver.cpp:106] Iteration 6800, lr = 0.0067767
10510 20:25:33.662945 15580 solver.cpp:228] Iteration 6900, loss = 0.0157094
10510 20:25:33.663949 15580 solver.cpp:244] Train net output #0: loss = 0.0157096 (* 1 = 0.0157096 loss)
10510 20:25:33.666962 15580 sgd_solver.cpp:106] Iteration 6900, lr = 0.0067466
10510 20:25:33.636912 15580 solver.cpp:327] Iteration 7000, Testing net (#0)
10510 20:25:41.681306 15580 solver.cpp:404] Test net output #0: accuracy = 0.9909
10510 20:25:41.681306 15580 solver.cpp:404] Test net output #1: loss = 0.0279062 (* 1 = 0.0279062 loss)
搜狗拼音输入法 全 :81 15580 solver.cpp:404]

```

图 3.6 手写数字识别案例运行后显示的训练和测试信息

```

st/lenet_iter_10000.caffemodel
10510 20:28:23.836871 15580 sgd_solver.cpp:273] Snapshotting solver state to binary proto file C:\Caffe\caffe-windows\mnist-examples/mnist/lenet_iter_10000.solverstate
10510 20:28:23.871765 15580 solver.cpp:317] Iteration 10000, loss = 0.0076271
10510 20:28:23.872776 15580 solver.cpp:327] Iteration 10000, Testing net (#0)
10510 20:28:27.010114 15580 solver.cpp:404] Test net output #0: accuracy = 0.9917
10510 20:28:27.011152 15580 solver.cpp:404] Test net output #1: loss = 0.0267212 (* 1 = 0.0267212 loss)
10510 20:28:27.012152 15580 solver.cpp:822] Optimization Done.
10510 20:28:27.012152 15580 caffe.cpp:282] Optimization Done.
C:\Caffe\caffe-windows\mnist\bin>

```

图 3.7 手写数字识别案例训练完成时的最终结果



3.6 LeNet 的交通标志识别案例

本节描述一个利用 LeNet 对交通标志进行识别的案例，其中用到的 GTSRB 数据集可以根据表 1.2 提供的地址下载，并需要按照有关说明进行选择和处理。

3.6.1 交通标志数据集的格式转换

GTSRB 数据集的图像是 .jpg 格式。由于 .jpg 格式的图像无法被 Caffe 直接使用，所以需要先把训练集和测试集转换为 Caffe 专有的 LEVELDB 格式类型。下面说明数据格式转换的具体步骤。

首先，把所有 .jpg 训练图像按类别目录组织拷贝到 ./caffe-windows-master/examples/mydata/train/ 目录下（如图 3.8 和图 3.9 所示），把所有 .jpg 测试图像直接拷贝到 ./caffe-windows-master/examples/mydata/test/ 目录下（如图 3.10 所示）。

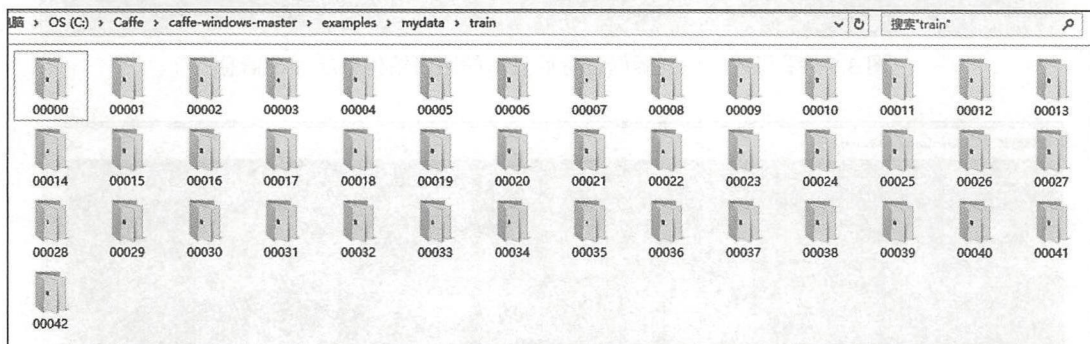


图 3.8 GTSRB 的训练集类别目录，共 43 个

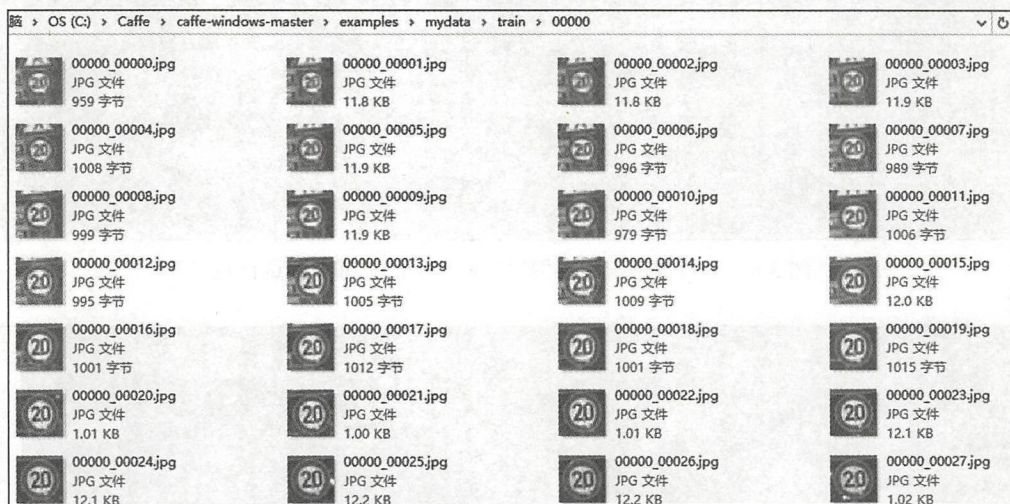


图 3.9 GTSRB 的训练集中的第 00000 类样本



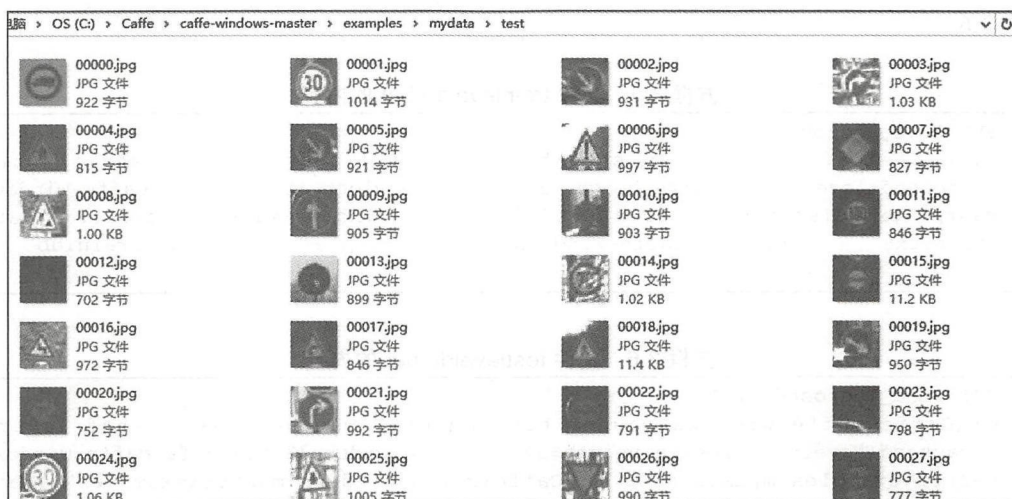


图 3.10 GTSRB 的测试集的部分样本

然后，在 C:\Caffe\caffe-windows-master\examples\mydata 目录下创建训练集的标签文件 train.txt 和测试集的标签文件 test.txt。train.txt 由所有训练图像的目录、名称及标签构成，其中每行的文本格式为“/ 图像目录 / 图像名称 标签”，样例如方框 3.3 所示。类似地，test.txt 由所有测试图像的名称及标签构成，每行的文本格式为“/ 图像名称 标签”，样例如方框 3.4 所示。

方框 3.3 标签文件 train.txt 的文本格式

```
/00000/00000_00000.jpg 0
/00000/00000_00001.jpg 0
/00000/00000_00002.jpg 0
/00000/00000_00003.jpg 0
/00000/00000_00004.jpg 0
```

方框 3.4 标签文件 test.txt 的文本格式

```
/00000.jpg 16
/00001.jpg 1
/00002.jpg 38
/00003.jpg 33
/00004.jpg 11
```

其次，用 convert_imageset.exe 命令把 GTSRB 数据集转换为 LEVELDB 格式。convert_imageset.exe 是 Caffe 自带的图像格式转换命令，存放在 C:\Caffe\caffe-windows-master\bin\ 目录下，可以把 jpg、png 和 jpeg 等常见格式转换成 LEVELDB 格式。具体的转换过程可以通过两个批处理文件完成：trainleveldb.bat 和 testleveldb.bat。文件的内容分别见方框 3.5 和



方框 3.6。

方框 3.5 文件 trainleveldb.bat 的内容

```
SET GLOG_logtostderr=1
C:\Caffe\caffe-windows-master\bin\convert_imageset.exe --shuffle=true
--backend=leveldb --resize_height=32 --resize_width=32 C:\Caffe\caffe-windows-
master\examples\mydata\train C:\Caffe\caffe-windows-master\examples\mydata\
train.txt C:\Caffe\caffe-windows-master\examples\mydata\ldb\trainldb
Pause
```

方框 3.6 文件 testleveldb.bat 的内容

```
SET GLOG_logtostderr=1
C:\Caffe\caffe-windows-master\bin\convert_imageset.exe --shuffle=true
--backend=leveldb --resize_height=32 --resize_width=32 C:\Caffe\caffe-windows-
master\examples\mydata\test C:\Caffe\caffe-windows-master\examples\mydata\
test.txt C:\Caffe\caffe-windows-master\examples\mydata\ldb\testldb
pause
```

在 trainleveldb.bat 和 testleveldb.bat 这两个文件中，“--shuffle=true”表示把转换后的数据集打乱，“--backend=leveldb”表示把数据转换为 LEVELDB 格式；“--resize_height=32 --resize_width=32”表示转换后图像的大小（这里由于保持大小不变，实际上可以省略）；“C:\Caffe\caffe-windows-master\examples\mydata\train”表示训练集所在目录；“C:\Caffe\caffe-windows-master\examples\mydata\train.txt”表示训练集的标签文件；“C:\Caffe\caffe-windows-master\examples\mydata\ldb\trainldb”表示转换后存放训练集的文件夹；“C:\Caffe\caffe-windows-master\examples\mydata\test”表示测试集所在目录；“C:\Caffe\caffe-windows-master\examples\mydata\test.txt”表示测试集的标签文件；“C:\Caffe\caffe-windows-master\examples\mydata\ldb\testldb”表示转换后存放测试集的文件夹。

最后，通过执行批处理文件 trainleveldb.bat 和 testleveldb.bat，就可以把训练集和测试集转换为 LEVELDB 格式，分别存放在 trainldb 和 testldb 文件夹下。

3.6.2 交通标志的识别分类

在建立好交通标志的训练集和测试集后，利用 LeNet 进行交通标志识别，只需根据表 3.7 和表 3.8 分别在 lenet_train_test.prototxt 和 lenet_solver.prototxt 文件中设置相应的参数值，再参照图 3.11 的命令运行即可。程序运行后，将依次出现一系列信息窗口界面，如图 3.12～图 3.14 所示。图 3.12 显示的是求解器参数配置信息，实质上就是加载文件 lenet_solver.prototxt 的内容。图 3.13 显示的是在训练网络的过程中，随着迭代次数的增加，学习率、损失函数、训练准确率和测试准确率的变化。在迭代训练 10 000 次后，程序运行结束，图 3.14 显示训练集的损失为 0.005 127 02，测试集的损失为 0.533 425，测试准确率为



91.50%。

表 3.7 交通标志识别案例在 lenet_train_test.prototxt 文件中设置的参数值

参数	设置值
CaseName	GTSRB
TrainSetDir	C:/Caffe/caffe-windows-master/examples/mydata/ldb/trainldb
Train_batch_size	64
TestSetDir	C:/Caffe/caffe-windows-master/examples/mydata/ldb/testldb
Train_batch_size	100
conv1_num_output	20
conv1_kernel_size	5
conv1_stride	1
pool1_kernel_size	2
pool1_stride	2
conv2_num_output	50
conv2_kernel_size	5
conv2_stride	1
pool2_kernel_size	2
pool2_stride	2
ip1_num_output	500
ip2_num_output	43

表 3.8 交通标志识别案例在 lenet_solver.prototxt 文件中设置的超参数值

超参数	设置值
lenet_train_test.prototxt_dir	C:/Caffe/caffe-windows-master/examples/mydata/lenet_train_test.prototxt
test_iter	100
test_interval	500
base_lr	0.01
momentum	0.9
weight_decay	0.000 5
lr_policy	inv
gamma	0.000 1
power	0.75
display	100
max_iter	10 000
snapshot	5 000
snapshot_prefix	C:/Caffe/caffe-windows-master/examples/mydata/lenet
solver_mode	CPU



```

选择C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation. 保留所有权利。

C:\Windows\system32>cd C:\Caffe\caffe-windows-master\bin

C:\Caffe\caffe-windows-master\bin>caffe train --solver="C:\Caffe\caffe-windows-master\examples\mydata\lenet_solver.prototxt"
I0515 16:09:40.975097 21943 caffe.cpp:178] Use CPU.

```

图 3.11 交通标志识别案例的运行命令

```

选择C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation. 保留所有权利。

C:\Windows\system32>cd C:\Caffe\caffe-windows-master\bin

C:\Caffe\caffe-windows-master\bin>caffe train --solver="C:\Caffe\caffe-windows-master\examples\mydata\lenet_solver.prototxt"
I0515 16:09:40.975097 21943 caffe.cpp:178] Use CPU.
I0515 16:09:40.977103 21943 solver.cpp:48] Initializing solver from parameters:
test_iter: 100
test_interval: 500
base_lr: 0.01
display: 100
max_iter: 10000
lr_policy: "inv"
gamma: 0.0001
power: 0.75
momentum: 0.9
weight_decay: 0.0005
snapshot: 5000
snapshot_prefix: "C:/Caffe/caffe-windows-master/examples/mydata/lenet"
solver_mode: CPU
net: "C:/Caffe/caffe-windows-master/examples/mydata/lenet_train_test.prototxt"
I0515 16:09:40.978106 21943 solver.cpp:91] Creating training net from net file: C:/Caffe/caffe-windows-master/examples/mydata/lenet_train_test.prototxt
I0515 16:09:40.979109 21943 net.cpp:313] The NetState phase (0) differed from the phase (1) specified by a rule in layer mnist
I0515 16:09:40.979109 21943 net.cpp:313] The NetState phase (0) differed from the phase (1) specified by a rule in layer accuracy
搜狗拼音输入法 全 :09 21943 net.cpp:49] Initializing net from parameters:

```

图 3.12 交通标志识别案例运行后显示的求解器参数配置信息

```

选择C:\Windows\system32\cmd.exe
I0515 16:10:04.033196 21943 solver.cpp:244] Train net output #0: loss = 1.22411 (* 1 = 1.22411 loss)
I0515 16:10:04.033196 21943 sgd_solver.cpp:106] Iteration 200, lr = 0.00985258
I0515 16:10:10.963399 21943 solver.cpp:228] Iteration 300, loss = 0.746801
I0515 16:10:10.964402 21943 solver.cpp:244] Train net output #0: loss = 0.746801 (* 1 = 0.746801 loss)
I0515 16:10:10.965405 21943 sgd_solver.cpp:106] Iteration 300, lr = 0.00978075
I0515 16:10:18.046273 21943 solver.cpp:228] Iteration 400, loss = 0.43527
I0515 16:10:18.047276 21943 solver.cpp:244] Train net output #0: loss = 0.43527 (* 1 = 0.43527 loss)
I0515 16:10:18.047276 21943 sgd_solver.cpp:106] Iteration 400, lr = 0.00971013
I0515 16:10:24.990978 21943 solver.cpp:337] Iteration 500, Testing net (#0)
I0515 16:10:30.351105 21943 solver.cpp:404] Test net output #0: accuracy = 0.8362
I0515 16:10:30.351105 21943 solver.cpp:404] Test net output #1: loss = 0.638114 (* 1 = 0.638114 loss)
I0515 16:10:30.427466 21943 solver.cpp:228] Iteration 500, loss = 0.345893
I0515 16:10:30.427963 21943 solver.cpp:244] Train net output #0: loss = 0.345893 (* 1 = 0.345893 loss)
I0515 16:10:30.428462 21943 sgd_solver.cpp:106] Iteration 500, lr = 0.00964069
I0515 16:10:37.370854 21943 solver.cpp:228] Iteration 600, loss = 0.307412
I0515 16:10:37.371857 21943 solver.cpp:244] Train net output #0: loss = 0.307412 (* 1 = 0.307412 loss)
I0515 16:10:37.371857 21943 sgd_solver.cpp:106] Iteration 600, lr = 0.0095724
I0515 16:10:44.365990 21943 solver.cpp:228] Iteration 700, loss = 0.204354
I0515 16:10:44.366982 21943 solver.cpp:244] Train net output #0: loss = 0.204354 (* 1 = 0.204354 loss)
I0515 16:10:44.366982 21943 sgd_solver.cpp:106] Iteration 700, lr = 0.00950522
I0515 16:10:51.307970 21943 solver.cpp:228] Iteration 800, loss = 0.239469
I0515 16:10:51.307970 21943 solver.cpp:244] Train net output #0: loss = 0.239469 (* 1 = 0.239469 loss)
I0515 16:10:51.308972 21943 sgd_solver.cpp:106] Iteration 800, lr = 0.00943913
I0515 16:10:58.203585 21943 solver.cpp:228] Iteration 900, loss = 0.122375
I0515 16:10:58.204587 21943 solver.cpp:244] Train net output #0: loss = 0.122375 (* 1 = 0.122375 loss)
I0515 16:10:58.204587 21943 sgd_solver.cpp:106] Iteration 900, lr = 0.00937411
I0515 16:11:05.143069 21943 solver.cpp:337] Iteration 1000, Testing net (#0)
I0515 16:11:10.520135 21943 solver.cpp:404] Test net output #0: accuracy = 0.8781
I0515 16:11:10.520135 21943 solver.cpp:404] Test net output #1: loss = 0.498095 (* 1 = 0.498095 loss)
搜狗拼音输入法 全 :90 21943 solver.cpp:228] Iteration 1000, loss = 0.174857

```

图 3.13 交通标志识别案例运行后显示的训练和测试信息


```

C:\Windows\system32\cmd.exe
I0515 16:22:41.963109 21948 solver.cpp:404] Test net output #0: accuracy = 0.9175
I0515 16:22:41.963644 21948 solver.cpp:404] Test net output #1: loss = 0.518224 (* 1 = 0.518224 loss)
I0515 16:22:42.038311 21948 solver.cpp:228] Iteration 9500, loss = 0.0227776
I0515 16:22:42.038810 21948 solver.cpp:244] Train net output #0: loss = 0.0227775 (* 1 = 0.0227775 loss)
I0515 16:22:42.038810 21948 sgd_solver.cpp:106] Iteration 9500, lr = 0.00606002
I0515 16:22:42.849491 21948 solver.cpp:228] Iteration 9600, loss = 0.0134931
I0515 16:22:42.849982 21948 solver.cpp:244] Train net output #0: loss = 0.013493 (* 1 = 0.013493 loss)
I0515 16:22:42.849982 21948 sgd_solver.cpp:106] Iteration 9600, lr = 0.00603682
I0515 16:22:42.849982 21948 solver.cpp:228] Iteration 9700, loss = 0.00470082
I0515 16:22:42.849982 21948 solver.cpp:244] Train net output #0: loss = 0.0047007 (* 1 = 0.0047007 loss)
I0515 16:22:42.849982 21948 sgd_solver.cpp:106] Iteration 9700, lr = 0.00601382
I0515 16:23:02.537005 21948 solver.cpp:228] Iteration 9800, loss = 0.0181822
I0515 16:23:02.537005 21948 solver.cpp:244] Train net output #0: loss = 0.0181821 (* 1 = 0.0181821 loss)
I0515 16:23:02.537005 21948 sgd_solver.cpp:106] Iteration 9800, lr = 0.00599102
I0515 16:23:02.537005 21948 solver.cpp:228] Iteration 9900, loss = 0.0070401
I0515 16:23:02.537005 21948 solver.cpp:244] Train net output #0: loss = 0.00703998 (* 1 = 0.00703998 loss)
I0515 16:23:02.537005 21948 sgd_solver.cpp:106] Iteration 9900, lr = 0.00596343
I0515 16:23:16.436761 21948 solver.cpp:454] Snapshotting to binary proto file C:/Caffe/caffe-windows-master/examples/mydata/lenet_iter_10000.caffemodel
I0515 16:23:16.494796 21948 sgd_solver.cpp:273] Snapshotting solver state to binary proto file C:/Caffe/caffe-windows-master/examples/mydata/lenet_iter_10000.solverstate
I0515 16:23:16.542420 21948 solver.cpp:317] Iteration 10000, loss = 0.00512702
I0515 16:23:16.542922 21948 solver.cpp:337] Iteration 10000, Testing net (#0)
I0515 16:23:22.435673 21948 solver.cpp:404] Test net output #0: accuracy = 0.915
I0515 16:23:22.436174 21948 solver.cpp:404] Test net output #1: loss = 0.533425 (* 1 = 0.533425 loss)
I0515 16:23:22.436174 21948 solver.cpp:322] Optimization Done.
I0515 16:23:22.436676 21948 caffe.cpp:222] Optimization Done.

C:\Caffe\caffe-windows-master\bin>
搜狗拼音输入法 全：

```

图 3.14 交通标志识别案的最终运行结果

3.7 LeNet 的交通路网提取案例

本节描述一个从遥感图像中自动提取交通路网的案例，其中用到的 RRSI 数据集可以根据表 1.2 提供的地址下载。交通路网是指公路、城市道路和单位管辖范围允许社会机动车通行的地方，包括广场、公共停车场等用于公众通行的场所。本案例要解决的问题是：给出一幅交通路网的遥感图像，如图 3.15a 所示，通过一系列的算法和程序实现自动提取路网的目的，如图 3.15b 所示。

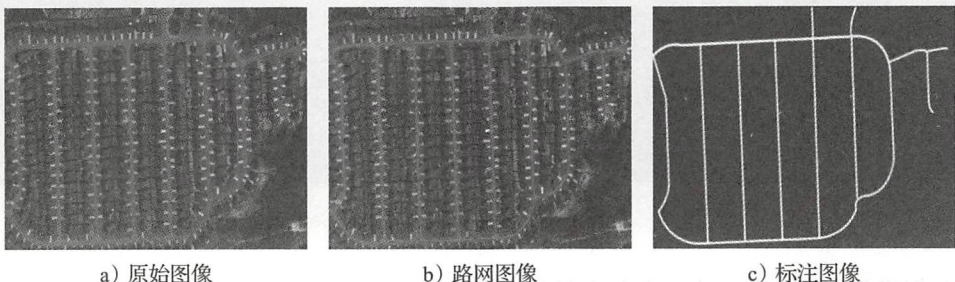


图 3.15 交通路网遥感图像样例

为了解决这个问题，本案例的思路是：首先选择一部分如图 3.15a 所示的原始图像，利用 ArcGis（或 ENVI、Photoshop 等其他软件工具）进行人工标注路网，另存为如图 3.15c 所示的标注图像。然后，把交通路网的自动提取问题转化为图像块的局部分类问题，转化的方法是把原始图像分解为一系列较小的图像块，如果图像块的中心在路网上，就标记为正

例（用 1 表示），否则就把偏离路网超过一定距离的图像块标记为反例（用 0 表示），从而就可以把路网看作正例图像块中心的集合。其次，挑选部分正例和反例训练一个 LeNet 分类模型。再次，利用这个模型对测试图像的所有 25×25 图像块进行分类，把正例标注为路网颜色，得到路网提取的初步结果。最后，利用连通性对正例进一步处理，形成路网提取的最终结果。

3.7.1 交通路网的人工标注

在原始遥感图像中人工标注交通路网，可以利用很多不同的软件工具，例如 Adobe Photoshop。Photoshop 是一款出色的数字图像编辑处理工具，进行路网标注的具体示例过程如下。

1) 导入图像。打开 Photoshop，单击选择需要标注的图像，拖入工作界面，如图 3.16 所示。



图 3.16 导入遥感图像

2) 创建新图层。单击右下角红色按钮，创建两个新图层，如图 3.17 所示。

3) 设置图层 1 的填充色。先选择图层 1，把前景色设置为黑色，再从顶部菜单中单击“编辑”，选择“填充”，弹出如图 3.18 所示的对话框，单击“确定”按钮即可，结果如图 3.19 所示。

4) 关闭图层 1 的可见性。选择图层 2，参照图 3.20 单击红色框中的按钮，即可关闭图层 1 的“指示图层可见性”。

5) 标注路网段。在左边工具条中选择“钢笔工具”，从路网上的任意点开始，沿着路网重复执行单击和移动操作，根据路网的弯曲调整单击的密度，标注一段路径后停止，设置前景色为白色，在该路径上右键选择“描边路径”，出现如图 3.21 所示的对话框，单击“确定”按钮，得到路网段标注的初步结果，如图 3.22 所示。随后，在该路径上右键选择“删除路径”，得到路网段标注的完成结果，如图 3.23 所示。

6) 标注全路网。重复第 5 步的操作标注其他路网段，得到的全路网标注图像如图 3.24 所示。再选择图层 1，单击“指示图层可见性”按钮，即可得到去除背景的全路网标注结果，如图 3.25 所示。保存结果。注意，这里要求标注图像在保存时的命名与原始图像相同。

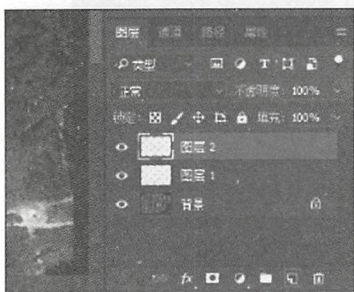


图 3.17 创建新图层

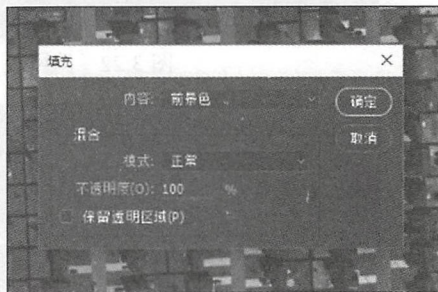


图 3.18 图层 1 填充命令窗口

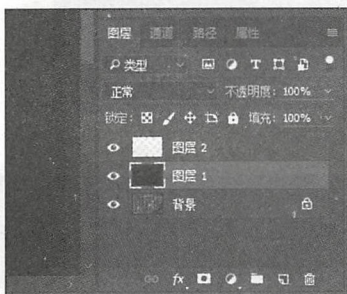


图 3.19 图层 1 填充结果

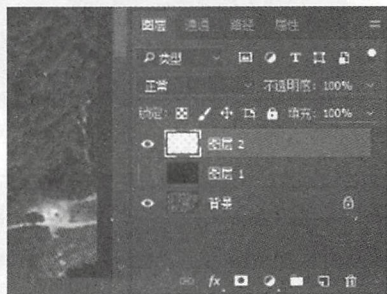


图 3.20 图层 1 的“指示图层可见性”按钮

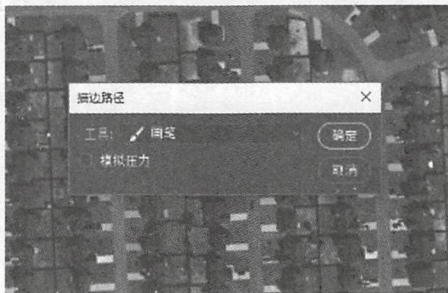


图 3.21 “描边路径”命令界面

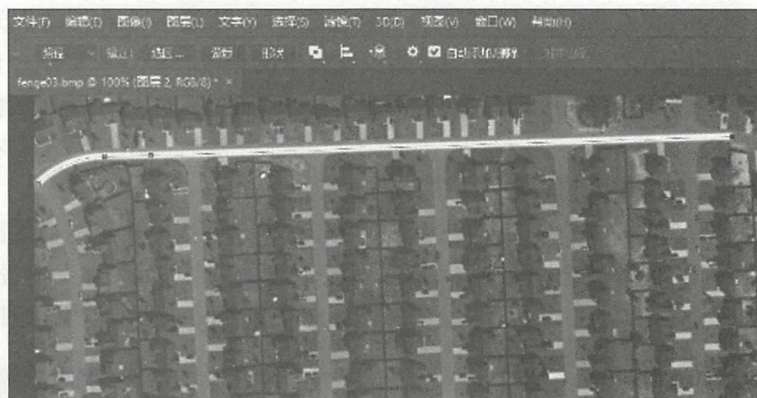


图 3.22 路网段标注的初步结果

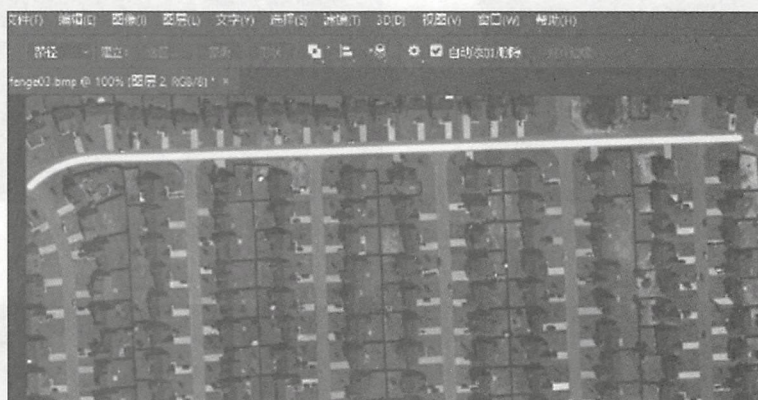


图 3.23 路网段标注的完成结果

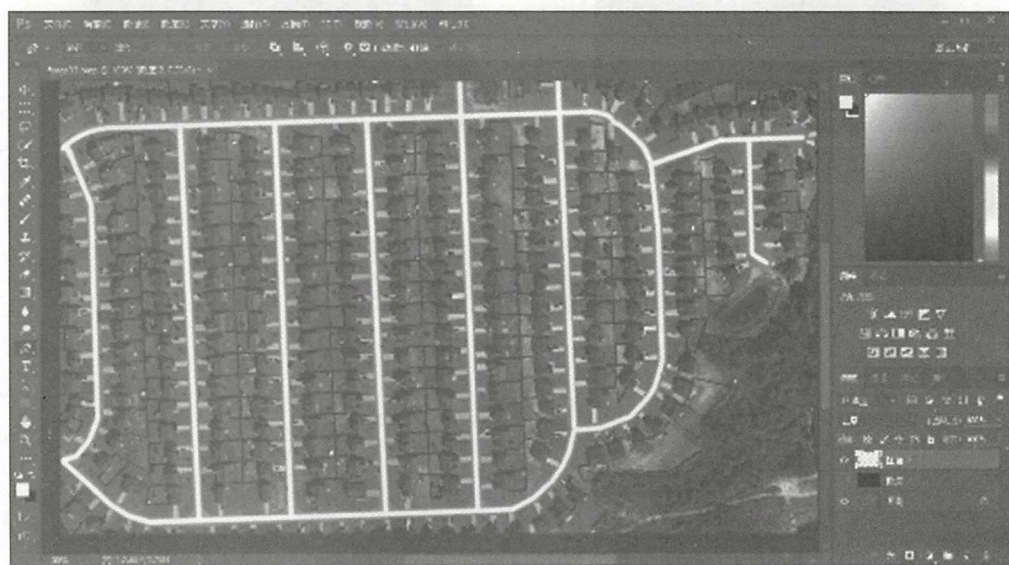


图 3.24 全路网标注图像

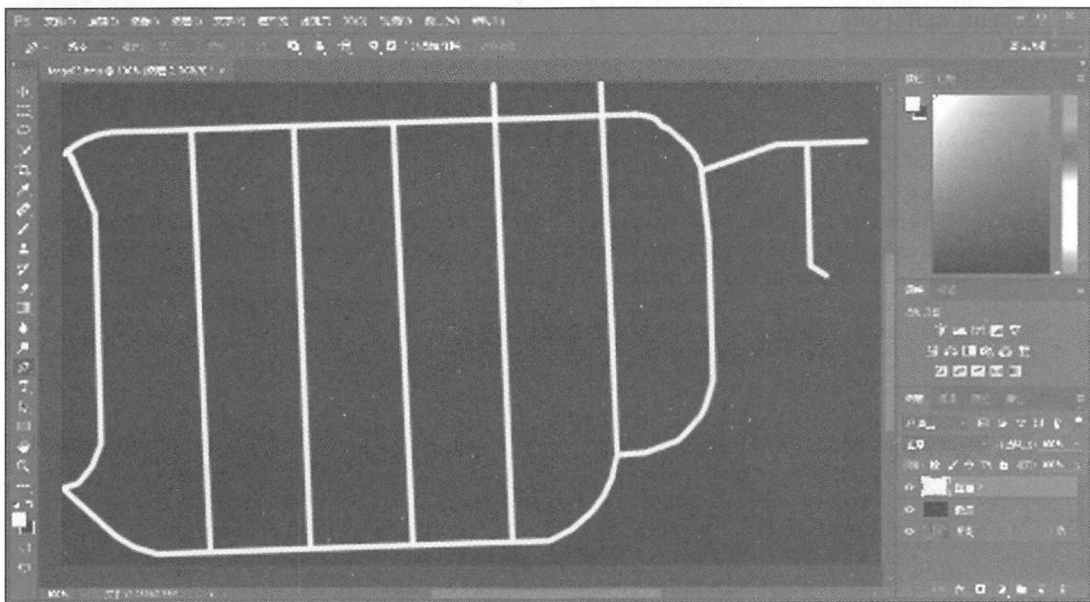


图 3.25 去除背景的全路网标注结果

3.7.2 交通路网的图像块分类

本案例把交通路网的自动提取问题转化为图像块的局部分类问题，转化的方法是把原始图像分解为一系列较小的图像块，如果图像块的中心在路网上，就标记为正例（用 1 表示），否则就标记为反例（用 0 表示），从而就可以把路网看作正例图像块中心的集合。基于这种局部分类的思路，本小节把 11 幅训练遥感图像，按每幅从左到右、从上到下的顺序扫描，抽取所有 25×25 大小的图像块，并进行正反例标注。对于一幅大小为 $m \times n$ 的图像，共可抽取 $(m-24)(n-24)$ 个图像块，其中可能包含数目不等的正例和反例。为了防止正例和反例的数目差别太大，需要对它们的比例进行控制，本案例控制比例约为 1:9。下面是抽取图像块的 Matlab 程序（sample.m 和 scan.m）和构造训练测试集的 Python 程序（getImageList.py）。

1. sample.m 的代码

```
clc;clear;
imageDir = './images/trainImages/';           % 训练图像的存放目录
labelDir = './images/labels/';               % 训练图像的对应标注图像所在目录
savePath = './data/';                       % 正反例图像块的存放目录
labelFile = dir(fullfile(labelDir,'*.bmp'));
wdSize = {[25,25]};                         % 图像块的大小
for i=1:length(labelFile)
    L = imread(fullfile(labelDir,labelFile(i).name)); % 读取第 i 个标注图像
    I = imread(fullfile(imageDir,labelFile(i).name)); % 读取第 i 个训练图像
    disp(['process:',labelDir,labelFile(i).name]);
    scan(L,I,wdSize,savePath);               % 调用 scan 函数抽取图像块
end
```

2. scan.m 的代码

```
function [ ] = scan(L,I,wdSize,savePath)
    % L 是标注图像, I 是原始图像, wdSize 是扫描窗口大小
    L = L(:,:,1); % 读取标注图像的 R 通道, 若要读取 G(B) 通道, 把“1”改为“2(3)”
    l_h = size(L,1); % 原始图像的高
    l_w = size(L,2); % 原始图像的宽
    p_num = length(dir(fullfile(savePath,'1/','*.bmp'))); % 当前存储的正例图像块数量
    n_num = length(dir(fullfile(savePath,'0/','*.bmp'))); % 当前存储的反例图像块数量
    disp([num2str(p_num),' | ',num2str(n_num)]);
    wd_h = wdSize{1}(1); % wdSize = {[25,25]}, wd_h 为扫描窗口的高
    wd_w = wdSize{1}(2); % wd_w 为扫描窗口的宽
    for i=1:l_h-wd_h
        for j=1:l_w-wd_w
            patch = L(i:i+wd_h,j:j+wd_w); % 抽取标注图像块
            if patch(13,13)==255 % 判断 patch 的中心是否在路网上
                p_num = p_num + 1; % 增加正例图像块的统计计数
                obj = I(i:i+wd_h-1,j:j+wd_w-1,:); % 抽取原始图像块
                imwrite(obj,[savePath,'1/','1_',num2str(p_num),'.bmp']);
                % 保存在 ./data/1/ 文件夹下
            elseif length(find(patch(14:22,14:22)==255))==0
                % patch 中心的 9×9 区域无路网穿过
                n_num = n_num + 1; % 增加反例图像块的统计计数
                rnd = randi(80,1,1); % 生成一个 1~80 的随机整数
                if rnd==1; % 随机整数个数等于 1, 存储该反例, 相当于按 1/80 概率抽取反例
                    bcg = I(i:i+wd_h-1,j:j+wd_w-1,:);
                    imwrite(bcg,[savePath,'0/','0_',num2str(n_num),'.bmp']);
                    % 保存在 ./data/0/ 文件夹下
                end
            end
        end
    end
end
end
end
```

3. getImageList.py 的代码

```
#!/user/bin/python
#!/conding=utf8

import os
import random
root = 'data'
dir = os.listdir(root) # 根目录
l = 0;
filelist = []; # 空列表
for d in dir: # 用 d 遍历 dir 下的文件夹, 这里仅包括 0 和 1 文件夹
    files = os.listdir(root+'/'+d) # files 指向 data 里的 0 或者 1 文件夹
    for f in files:
        filelist.append('/'+root + '/' + d + '/' + f + ' ' + d + '\n')
        # 在文件名后加标签
    l = l + 1
random.shuffle(filelist) # 随机打乱 filelist 中的排序
```

```

txt = file('imglist_train.txt', 'w+')    # 准备写入
txt.writelines(filelist[0:int(len(filelist)*0.90)]) # 选择 90% 带标签的图像块构造训练集
txt.close()
txt = file('imglist_test.txt', 'w+')
txt.writelines(filelist[int(len(filelist)*0.90):len(filelist)])
                                     # 选择剩余的 10% 构造测试集
txt.close()

```

依次运行程序 sample.m、scan.m 和 getImageList.py，就可以得到局部分类的图像块训练集和测试集，有关路径及标签分别保存在 imglist_train.txt 和 imglist_test.txt 这两个文件中。其中，训练集和测试集按照 9:1 的比例划分，训练集包含 121 257 幅图像，测试集包含 13 474 幅图像。

方框 3.7 批处理文件 trainleveldb.bat 的内容

```

SET GLOG_logtostderr=1
convert_imageset.exe --backend=leveldb ./ ./imglist_train.txt ./ldb/trainldb
pause

```

方框 3.8 批处理文件 testleveldb.bat 的内容

```

SET GLOG_logtostderr=1
convert_imageset.exe --backend=leveldb ./ ./imglist_test.txt ./ldb/testldb
pause

```

为了得到 Caffe 能够识别的训练集和测试集，还需要参照 3.6.1 节利用 convert_imageset.exe 把 imglist_train.txt 和 imglist_test.txt 中涉及的数据转换成 LEVELDB 格式。具体的转换过程可以通过两个批处理文件完成：trainleveldb.bat 和 testleveldb.bat。文件的内容分别见方框 3.7 和方框 3.8。

3.7.3 交通路网的图像块分类 LeNet

在建立好局部图像块的正反例训练集和测试集后，利用 LeNet 对这些局部图像块进行分类，只需按照表 3.9 和表 3.10 分别在 lenet_train_test.prototxt 和 lenet_solver.prototxt 文件中设置相应的参数值，再参照图 3.26 的命令运行。在迭代训练 200 000 次后，程序运行结束，图 3.27 显示训练集的损失为 0.001 550 54，测试集的损失为 0.006 925 75，测试准确率为 99.80%。按照 Caffe 的命名规则，训练好的权值和偏置存放在 _iter_200000.caffemodel 文件中。

表 3.9 图像块分类 LeNet 程序在 lenet_train_test.prototxt 文件中设置的参数值

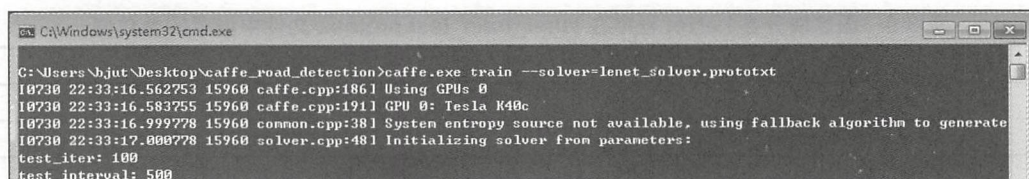
参数	设置值
CaseName	TRNE
TrainSetDir	./ldb/trainldb
Train_batch_size	128
TestSetDir	./ldb/testldb

(续)

参数	设置值
Train_batch_size	100
conv1_num_output	64
conv1_kernel_size	5
conv1_stride	1
pool1_kernel_size	2
pool1_stride	2
conv2_num_output	32
conv2_kernel_size	3
conv2_stride	1
pool2_kernel_size	2
pool2_stride	2
ip1_num_output	64
ip2_num_output	2

表 3.10 图像块分类 LeNet 程序在 lenet_solver.prototxt 文件中设置的超参数值

超参数	设置值
lenet_train_test.prototxt_dir	./lenet_train_test.prototxt
test_iter	100
test_interval	500
base_lr	0.01
momentum	0.9
weight_decay	0.000 5
lr_policy	inv
gamma	0.000 1
power	0.75
display	100
max_iter	200 000
Snapshot	1 000
snapshot_prefix	./snapshots/
solver_mode	GPU



```

C:\Windows\system32\cmd.exe
C:\Users\hjut\Desktop>caffe.exe train --solver=lenet_solver.prototxt
10730 22:33:16.562753 15960 caffe.cpp:1861 Using GPUs 0
10730 22:33:16.583755 15960 caffe.cpp:1911 GPU 0: Tesla K40c
10730 22:33:16.999778 15960 common.cpp:381 System entropy source not available, using fallback algorithm to generate
10730 22:33:17.000778 15960 solver.cpp:481 Initializing solver from parameters:
test_iter: 100
test_interval: 500

```

图 3.26 图像块分类 LeNet 程序的运行命令

```

C:\Windows\system32\cmd.exe
10730 23:37:51.026360 15116 sgd_solver.cpp:1061 Iteration 199200, lr = 0.0010223
10730 23:37:52.823463 15116 solver.cpp:2281 Iteration 199300, loss = 0.00281335
10730 23:37:52.824463 15116 solver.cpp:2441 Train net output #0: loss = 0.00281325 (* 1 = 0.00281325 loss)
10730 23:37:52.826463 15116 sgd_solver.cpp:1061 Iteration 199300, lr = 0.00102193
10730 23:37:54.665568 15116 solver.cpp:2281 Iteration 199400, loss = 0.00345953
10730 23:37:54.665568 15116 solver.cpp:2441 Train net output #0: loss = 0.00345943 (* 1 = 0.00345943 loss)
10730 23:37:54.667568 15116 sgd_solver.cpp:1061 Iteration 199400, lr = 0.00102157
10730 23:37:56.203656 15116 solver.cpp:3371 Iteration 199500, Testing net #0
10730 23:37:57.139710 15116 solver.cpp:4041 Test net output #0: accuracy = 0.9973
10730 23:37:57.139710 15116 solver.cpp:4041 Test net output #1: loss = 0.00860627 (* 1 = 0.00860627 loss)
10730 23:37:57.143710 15116 solver.cpp:2281 Iteration 199500, loss = 0.009991974
10730 23:37:57.144711 15116 solver.cpp:2441 Train net output #0: loss = 0.009991872 (* 1 = 0.009991872 loss)
10730 23:37:57.146710 15116 sgd_solver.cpp:1061 Iteration 199500, lr = 0.0010212
10730 23:37:58.604794 15116 solver.cpp:2281 Iteration 199600, loss = 0.00435333
10730 23:37:58.604794 15116 solver.cpp:2441 Train net output #0: loss = 0.00435323 (* 1 = 0.00435323 loss)
10730 23:37:58.607795 15116 sgd_solver.cpp:1061 Iteration 199600, lr = 0.00102084
10730 23:38:00.075878 15116 solver.cpp:2281 Iteration 199700, loss = 0.00382111
10730 23:38:00.075878 15116 solver.cpp:2441 Train net output #0: loss = 0.00382101 (* 1 = 0.00382101 loss)
10730 23:38:00.076878 15116 sgd_solver.cpp:1061 Iteration 199700, lr = 0.00102047
10730 23:38:01.686970 15116 solver.cpp:2281 Iteration 199800, loss = 0.00187064
10730 23:38:01.687970 15116 solver.cpp:2441 Train net output #0: loss = 0.00187054 (* 1 = 0.00187054 loss)
10730 23:38:01.689970 15116 sgd_solver.cpp:1061 Iteration 199800, lr = 0.00102011
10730 23:38:03.805091 15116 solver.cpp:2281 Iteration 199900, loss = 0.00129232
10730 23:38:03.806092 15116 solver.cpp:2441 Train net output #0: loss = 0.00129222 (* 1 = 0.00129222 loss)
10730 23:38:03.808091 15116 sgd_solver.cpp:1061 Iteration 199900, lr = 0.00101974
10730 23:38:05.326179 15116 solver.cpp:4541 Snapshotting to binary proto file ./snapshots/_iter_200000.caffemodel
10730 23:38:05.332178 15116 sgd_solver.cpp:2731 Snapshotting solver state to binary proto file ./snapshots/_iter_200
10730 23:38:05.335178 15116 solver.cpp:3171 Iteration 200000, loss = 0.00155054
10730 23:38:05.335178 15116 solver.cpp:3371 Iteration 200000, Testing net #0
10730 23:38:06.342236 15116 solver.cpp:4041 Test net output #0: accuracy = 0.998
10730 23:38:06.343236 15116 solver.cpp:4041 Test net output #1: loss = 0.00692575 (* 1 = 0.00692575 loss)
10730 23:38:06.345237 15116 solver.cpp:3221 Optimization Done.
10730 23:38:06.347236 15116 caffe.cpp:2231 Optimization Done.

G:\Users\hjut\Desktop\caffe_road_detection>

```

图 3.27 图像块分类 LeNet 程序的最终运行结果

3.7.4 交通路网的自动提取代码及说明

在图像块分类 LeNet 训练好后, 接下来就可以利用这个网络从遥感图像中自动提取交通路网。提取过程需要用到 lenet.prototxt 文件和 3 个 Python 程序。其中, lenet.prototxt 文件用来根据表 3.11 设置 LeNet 的结构, 通过结合已经训练好的权值和偏置, 对图像块进行分类。

表 3.11 图像块分类 LeNet 程序在 lenet.prototxt 文件中设置的参数值

参数	设置值
dim1、dim2、dim3、dim3、dim4	1、3、25、25
conv1_num_output	64
conv1_kernel_size	5
conv1_stride	1
pool1_kernel_size	2
pool1_stride	2
conv2_num_output	32
conv2_kernel_size	3
conv2_stride	1
pool2_kernel_size	2
pool2_stride	2
ip1_num_output	64
ip2_num_output	2

3 个 Python 程序分别是 `batch_classify.py`、`batch_remove_scattered_point.py` 和 `draw_points.py`。其中，`batch_classify.py` 调用训练好的 LeNet 对待提路网遥感图像中的所有图像块进行分类，如果是正例，就把中心标注为红色，否则保持原色，得到初步的路网提取结果。`batch_remove_scattered_point.py` 用来去除较小的红色连通区域。`draw_points.py` 则用来显示路网提取的最终结果。

下面是 `batch_classify.py`、`batch_remove_scattered_point.py` 和 `draw_points.py` 的代码及说明。

1. `batch_classify.py` 的代码及说明

```
import os
import sys
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from PIL import Image, ImageDraw
import caffe

def label_image(dir, image_file):
    image = Image.open(dir+image_file)
    h, w = image.size
    wdSize = [25, 25]
    plist = []
    for i in range(0, h-wdSize[1], 1):
        for j in range(0, w-wdSize[0], 1):
            region = (i, j, i+wdSize[0], j+wdSize[1])
            roi = image.crop(region)
            roi = np.array(roi, dtype='float32') # 转化为矩阵
            roi = roi/255 # 归一化
            prediction = net.predict([roi], oversample = False) # 预测图像块的类别
            if 0 != prediction[0].argmax(): # 如果不是反例，添加正例图像块中心坐标到 plist
                plist.append([i+int((wdSize[0]+1)/2), j+int((wdSize[1]+1)/2)])
    for i in range(len(plist)):
        draw = ImageDraw.Draw(image)
        x, y = plist[i]
        draw.arc((x-1, y-1, x+1, y+1), 0, 360, fill=255)
    image.save(r'./resultImages/'+image_file) # 保存带正例中心点的结果图像
    return plist

def write_list(my_list, file_name):
    file = open(file_name, 'w')
    for e in my_list:
        file.writelines(str(e[0])+' '+str(e[1])+'\n')
    file.close()

model_file=r'./LeNet.prototxt' # 提取网络模型框架
trained_model=r'./snapshots/_iter_200000.caffemodel' # 提取训练 200 000 次的模型参数
imageDir = r'./images/testImages/' # 提取测试图像目录
net = caffe.Classifier(model_file, trained_model, channel_swap=(2, 1, 0))
caffe.set_mode_gpu()
```



```

files = os.listdir(imageDir)                # 提取测试图像目录下的所有文件
for image_file in files:                    # 对每个图像文件提取正例中心并存储到相应文件
    plist = label_image(imageDir, image_file)
    name, suffix = image_file.split('.')
    write_list(plist, r'./resultPoints/'+name+r'.txt')

```

注意：在上述代码中，`net.predict` 是 Caffe 自带函数，用来对未知样本进行分类。

2. batch_remove_scattered_point.py 的代码及说明

```

import os
import numpy as np
import scipy.ndimage as ndi
from skimage import measure, color
import matplotlib.pyplot as plt

def read_file(filename):                    # 该函数用来读取正例中心坐标
    points = []
    with open(filename, 'r') as f:
        for line in f.readlines():
            linestr = line.strip()          # strip() 方法用于移除字符串头尾指定的字符（默
                                            # 认为空格）
            x, y = linestr.split(' ')
            print x, y
            points.append([int(x), int(y)])
    return points

def write_list(mylist, filename):            # 该函数用来存储正例中心坐标
    f = open(filename, 'w')
    for p in mylist:
        f.writelines(str(p[0])+' '+str(p[1])+'\n')
    f.close()

def get_adj(points, wdSize):                # 该函数用来生成一个关于正例中心的二值矩阵
    adj = np.zeros(wdSize)
    for p in points:
        adj[p[0], p[1]] = 1                # 将矩阵 adj 位置为 p[0], p[1] 的值为 1
    return adj

def adj_to_coordinate(adj):                 # 把二值矩阵的正例中心点转换成坐标数据
    points = []
    row, col = adj.shape                   # 把 adj 的宽和高赋给 row, col
    for i in range(row):
        for j in range(col):
            if adj[i, j] == 1:
                points.append([i, j])      # 把 [i, j] 添加到 points 串中
    return points

def get_labels(adj):                        # 标记 8 连通区域，并按顺序编号
    labels = measure.label(adj, connectivity=2) # 标记 8 连通区域
    print('regions number:', labels.max()+1)    # 显示连通区域的个数
    return labels

def zeros_elements(idx, image):             # 该函数用来将图像的某个区域置为 0
    for i in idx:
        image[i[0], i[1]] = 0

def remove_unroad(image, label_image, threshold): # 该函数用来去除小连通区域

```

```

rp = measure.regionprops(label_image)          # 检测连通区域的属性
for i in range(len(rp)):
    flag = False
    if rp[i].area<threshold:                    # 标注面积小于阈值的区域
        flag = True
    if flag == True:
        idx = np.argwhere(label_image == i+1)  # 统计小区域的点集
        zeros_elements(idx,image)             # 将小区域的点集置零
def max_points(plist):                          # 该函数用来计算点列的横纵坐标的最大值
    maxp = [0,0];
    for p in plist:
        if p[0]>maxp[0]:
            maxp[0] = p[0]
        if p[1]>maxp[1]:
            maxp[1] = p[1]
    return maxp
points_dir = './resultPoints/'                 # 初始正例中心坐标目录
save_dir = './resultPointsFinal/'             # 优化后的正例中心坐标目录
pfiles = os.listdir(points_dir)
wdSize = [0,0]
for pf in pfiles:
    points = read_file(points_dir+pf)
    if 0 == len(points):
        continue
    wdSize = max_points(points)
    wdSize[0] = wdSize[0] + 1
    wdSize[1] = wdSize[1] + 1
    adj = get_adj(points,wdSize)                # 生成正例中心的二值矩阵
    labels = get_labels(adj)                   # 标记连通区域并按顺序编号
    remove_unroad(adj,labels,200)              # 去除少于200个点的连通区域
    new_points = adj_to_coordinate(adj)         # 把剩余的正例中心点转换成坐标数据
write_list(new_points,save_dir+pf)

```

3. draw_points.py 的代码及说明

```

import os
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from PIL import Image, ImageDraw
def read_file(filename):                        # 该函数用来按整数格式读取正例中心点的坐标数据
    points = [];
    with open(filename,'r') as f:
        for line in f.readlines():
            linestr = line.strip()
            x,y = linestr.split(' ')
            print x,y
            points.append([int(x),int(y)])
    return points
def write_labeled_img(points,image,filename):   # 该函数用来将正例中心点标注为红色
    for i in range(len(points)):
        draw = ImageDraw.Draw(image)

```

```

        x,y=points[i]
        image.putpixel([x,y],(255,0,0))
    image.save(filename)
points_dir = r'./ resultPointsFinal/' # 优化后的正例中心坐标目录
image_dir = r'./images/testImages/' # 测试图像目录
save_dir = r'./resultImgesFinal/' # 图像路网提取结果的保存目录
pfile = os.listdir(points_dir)
for pf in pfile:
    points = read_file(points_dir+pf)
    name,suffix = pf.split('.')
    image = Image.open(image_dir+name+'.bmp')
    write_labeled_img(points,image,save_dir+name+'.bmp')

```

3.7.5 交通路网的自动提取程序运行结果

参照表 3.11 设置好参数后, 执行 Python 程序 `batch_classify.py` 从测试图像提取路网, 命令如图 3.28 所示。命令执行完成后, 提取的初始正例中心坐标如图 3.29 所示, 提取的初始交通路网如图 3.30 所示。然后, 执行 `batch_remove_scattered_point.py` 程序去除正例中心点的小连通区域, 命令如图 3.31 所示, 运行结果如图 3.32 所示。最后, 执行 `draw_points.py` 显示交通路网的最终提取结果, 命令如图 3.33 所示, 结果如图 3.34 所示。

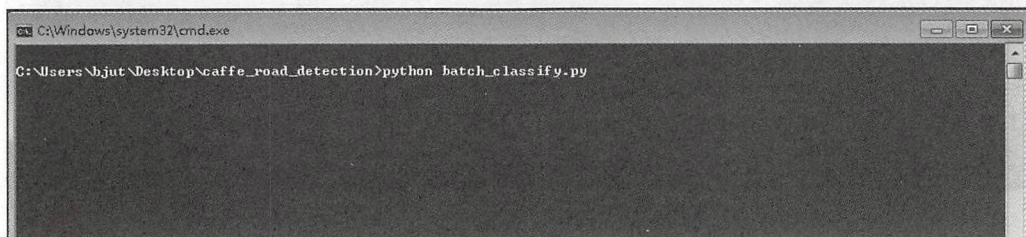


图 3.28 `batch_classify.py` 的运行命令

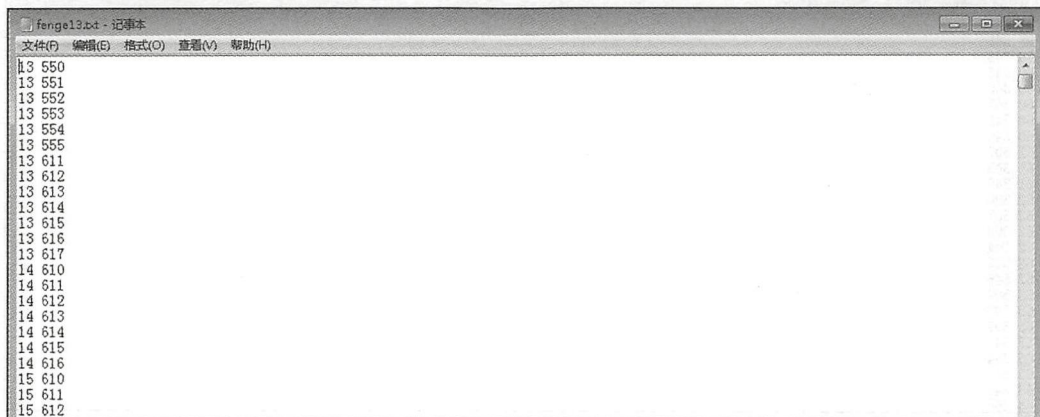


图 3.29 `batch_classify.py` 提取的初始正例中心坐标



图 3.30 batch_classify.py 提取的初始交通路网

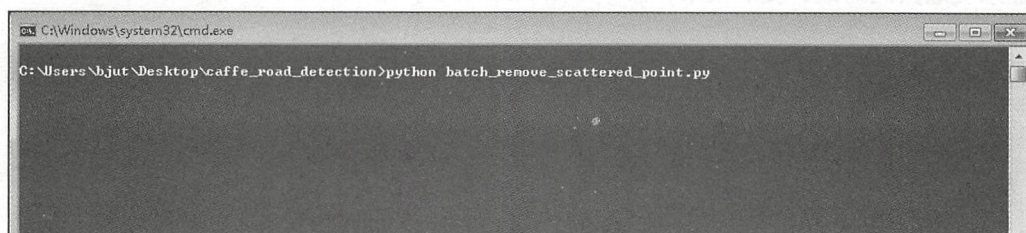


图 3.31 batch_remove_scattered_point.py 的执行命令

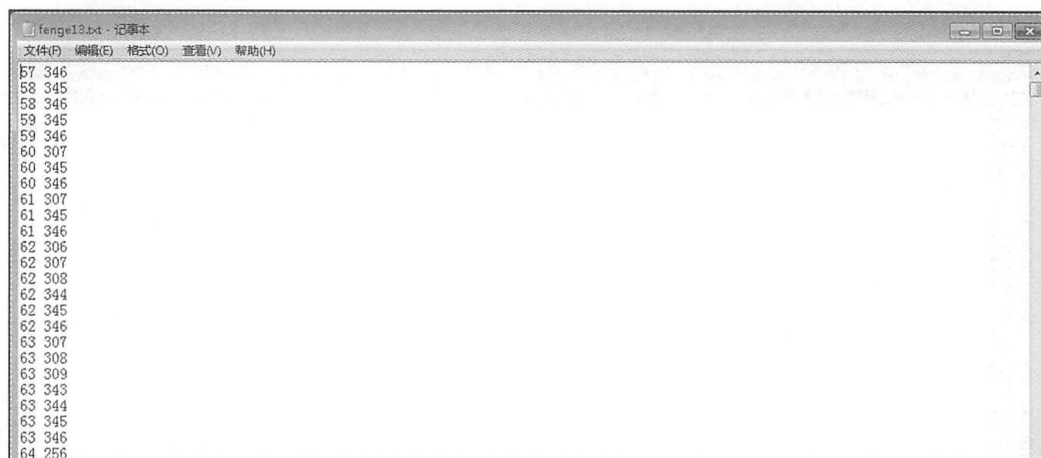


图 3.32 batch_remove_scattered_point.py 的运行结果

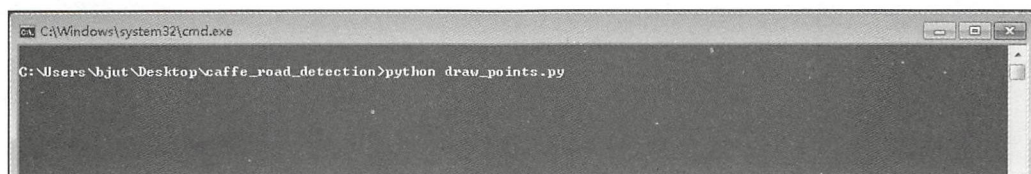


图 3.33 draw_points.py 的执行命令



图 3.34 交通路网案例的最终提取结果

卷积神经网络的突破模型

普遍认为深度学习的开端是 2006 年，但它受到学术界和工业界的广泛关注却是从 2012 年 AlexNet 在大规模图像分类中获得成功应用开始的。AlexNet 是深度学习发展史上的突破性成果，使神经网络重新回到了人工智能的风口浪尖。AlexNet 通过使用 GPU 显卡和校正线性单元（ReLU）极大地提高了卷积神经网络的学习训练速度，并在 2012 年的 ILSVRC 竞赛中取得了第一名的突出成绩。本章主要介绍 AlexNet 的模型结构、Caffe 和 TensorFlow 代码实现及说明，及其在 ImageNet 上的大规模图像分类案例及演示效果，并简要讨论其改进模型 ZFNet。

4.1 AlexNet 的模型结构

深度学习的快速发展是从 AlexNet 的提出及应用开始的。2010 年，斯坦福大学的李飞飞正式组织并启动了大规模视觉图像识别竞赛（ImageNet Large Scale Visual Recognition Challenge, ILSVRC）。ILSVRC 是一个关于图像、视频物体的识别分类竞赛平台，不仅极大地推动了计算机视觉发展，而且重新确立了神经网络，也就是现在的深度学习，在人工智能中的强大威力和领头地位。

在 ILSVRC 竞赛举办之前，机器学习用到的数据集大多相对较小，人们一般都喜欢用简单的模型和人工设计的特征来解决问题。在 ILSVRC 竞赛举办之后，ImageNet 作为一个很大的图像标注数据库开始对外开放，但深度学习技术并没有马上得到应用。事实上，在 2010 年和 2011 年的 ILSVRC 竞赛中，冠军获得者分别采用的方法是稀疏编码（sparse coding）和 SIFT + FV。其中，稀疏编码是一种模拟哺乳动物视觉系统初级皮层 V1 区简单细胞感受野的人工神经网络方法，具有空间的局部性、方向性和频域的带通性；SIFT 是指尺度不变特征变换（Scale-Invariant Feature Transform, SIFT），FV 是指 Fisher 向量（Fisher Vector）方法。2012 年，Alex Krizhevsky、Ilya Sutskever 和 Geoffrey E. Hinton 提出了一种非常重要的卷积神经网络模型^[109]，获得了 ILSVRC 图像分类的冠军，吸引了学术界和工业

界的广泛关注。这个模型的结构如图 4.1 所示，现在称为 AlexNet。

从图 4.1 可以看出，AlexNet 包含输入层、5 个卷积层和 3 个全连接层。其中，有 3 个卷积层还进行了最大池化。AlexNet 各层的组织结构如表 4.1 所示，其中 conv 表示卷积运算操作，ReLU 表示校正线性单元，pool 表示池化操作，norm 表示局部响应归一化，dropout 表示丢失输出操作，IP 表示全连接，softmax 表示软最大函数。

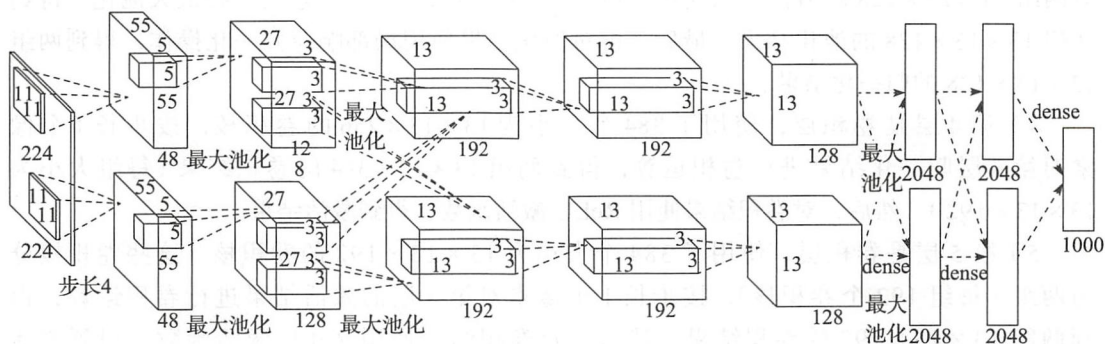


图 4.1 AlexNet 模型的结构示意图

表 4.1 AlexNet 的各层运算、维度和参数列表

	1	2	3	4	5	6	7	8
	Conv1	Conv2	Conv3	Conv4	Conv5	Fc6	Fc7	Fc8
	Data ↓ conv conv1	norm1 ↓ conv conv2	norm2 ↓ conv conv3	conv3 ↓ conv conv4	conv4 ↓ conv conv5	pool5 ↓ IP fc6	fc6 ↓ IP fc7	fc7 ↓ IP Fc8
Data	conv1 ↓ ReLU conv1	conv2 ↓ ReLU conv2	conv3 ↓ ReLU conv3	conv4 ↓ ReLU conv4	conv5 ↓ ReLU conv5	fc6 ↓ ReLU fc6	fc7 ↓ ReLU fc7	fc8 ↓ softmax prob
	conv1 ↓ pool pool1	conv2 ↓ pool pool2			conv5 ↓ pool pool5	fc6 ↓ dropout fc6	fc7 ↓ dropout fc7	prob ↓ label
	pool1 ↓ norm norm1	pool2 ↓ norm norm2						

下面是对 AlexNet 各层的详细描述。

1) 第 1 层是输入层，输入是大小为 224×224 的 3 通道图像。

2) 第 2 层是卷积层，使用了 96 个大小为 $11 \times 11 \times 3$ 的卷积核。这些卷积核分为两组（每组 48 个卷积核），按步长 4 个像素对输入层进行卷积运算，得到两组 $55 \times 55 \times 48$ 的卷积结果。然后，对卷积结果使用 ReLU 激活函数，得到激活结果。接着，对两组 $55 \times 55 \times 48$ 的激活结果使用窗口为 3×3 、步长为 2 个像素的重叠最大池化，得到两组 $27 \times 27 \times 48$ 的池

化结果。最后，再对池化结果使用局部响应归一化操作，得到两组 $27 \times 27 \times 48$ 的归一化结果。

3) 第3层是卷积层，使用了256个大小为 $27 \times 27 \times 48$ 的卷积核。这些卷积核分为两组（每组128个卷积核），按步长1个像素对第2层的归一化结果进行卷积运算，得到两组 $27 \times 27 \times 128$ 的卷积结果。然后，对卷积结果使用 ReLU 激活函数，得到激活结果。接着，对两组 $27 \times 27 \times 128$ 的激活结果使用窗口为 3×3 、步长为2个像素的重叠最大池化，得到两组 $13 \times 13 \times 128$ 的池化结果。最后，再对池化结果使用局部响应归一化操作，得到两组 $13 \times 13 \times 128$ 的归一化结果。

4) 第4层是卷积层，使用了384个大小为 $13 \times 13 \times 256$ 的卷积核，按步长1个像素对第3层归一化结果进行卷积运算，得到两组 $13 \times 13 \times 384$ 的卷积结果（每组大小为 $13 \times 13 \times 192$ ）。然后，对卷积结果使用 ReLU 激活函数，得到激活结果。

5) 第5层是卷积层，使用了384个大小为 $13 \times 13 \times 192$ 的卷积核。这些卷积核分为两组（每组192个卷积核），按步长1个像素对第4层的激活结果进行卷积运算，得到两组 $13 \times 13 \times 192$ 的卷积结果。然后，对卷积结果使用 ReLU 激活函数，得到激活结果。

6) 第6层是卷积层，使用了256个大小为 $13 \times 13 \times 192$ 的卷积核。这些卷积核分为两组（每组128个卷积核），按步长1个像素对第5层的激活结果进行卷积运算，得到两组 $13 \times 13 \times 128$ 的卷积结果。然后，对卷积结果使用 ReLU 激活函数，得到激活结果。接着，对两组 $13 \times 13 \times 128$ 的激活结果，进行窗口为 3×3 、步长为2个像素的重叠最大池化，得到两组 $13 \times 13 \times 128$ 的池化结果。

7) 第7层是全连接层，使用了4096个神经元，分为两组（每组2048个卷神经元），对第6层的池化结果进行全连接处理，然后对全连接处理结果使用 ReLU 激活函数，得到激活结果。接着，对激活结果使用概率为0.5的 dropout 操作，得到 dropout 结果。

8) 第8层是全连接层，使用了4096个神经元，分为两组（每组2048个卷神经元），对第7层的 dropout 结果进行全连接处理，然后对全连接处理结果使用 ReLU 激活函数，得到激活结果。接着，对激活结果使用概率为0.5的 dropout 操作，得到 dropout 结果。

9) 最后一层是1000路的软最大输出层，用来产生一个覆盖1000类的标签分布。

与 LeNet 相比，AlexNet 有了很多改进和优点，如表 4.2 所示。具体描述如下。

表 4.2 AlexNet 对比 LeNet 的主要新增训练技巧

技巧	作用	AlexNet	LeNet
ReLU、多个 GPU	提高训练速度	Y	N
重叠池化	提高精度、缓解过拟合	Y	N
局部响应归一化	提高精度	Y	N
数据扩充、丢失输出	减少过拟合	Y	N

1. 使用 ReLU 激活函数

在 AlexNet 之前, 神经网络 (包括 LeNet) 一般都把激活函数选为 sigmoid 或 tanh。这类函数在自变量非常大或者非常小时, 函数输出基本不变, 称为饱和函数。为了提高训练速度, AlexNet 使用了校正线性函数 $\text{ReLU}(x) = \max(0, x)$ 。ReLU 是一种非饱和函数, 在训练时间上比饱和函数更快。与 sigmoid 和 tanh 相比, ReLU 利用分片线性结构实现了非线性的表达能力, 梯度消失现象相对较弱, 有助于训练更深的网络。

2. 使用 GPU 训练

随着数据集越来越大, 机器学习对图形处理的计算力需求逐步超越了 CPU 的性能水平。这导致图形处理器 (Graphics Processing Unit, GPU) 得到了迅速发展。与 CPU 不同, GPU 是专为执行复杂的数学和几何计算而设计的。现在, GPU 已经超越了 3D 图形处理的局限, 被广泛应用于浮点运算和并行计算, 可以提供数十倍乃至上百倍于 CPU 的性能。早期的 LeNet 没有使用 GPU, 而 AlexNet 使用两个 GPU 来提升训练速度, 分别放置一半卷积核 (或神经元), 并限制在某些层之间进行 GPU 通信。

3. 局部响应归一化

为了改善卷积神经网络的性能, AlexNet 还对某些层进行了局部响应归一化处理。在理论上, 这种处理可以看作一个新增的层, 但在实际应用时一般不统计到总层数中。如果用 $a_{x,y}^i$ 表示第 i 个卷积面上在位置 (x, y) 的值, 则局部响应归一化的值 $b_{x,y}^i$ 是通过若干相邻卷积面在位置 (x, y) 的值来计算的, 公式如下:

$$b_{x,y}^i = a_{x,y}^i / (k + \alpha \sum_{j=\max(0, i-n/2)}^{\min(N-1, i+n/2)} (a_{x,y}^j)^2)^\beta \quad (4.1)$$

其中, N 是卷积面 (或池化面) 的总数, n 是相邻面的个数, k, α, β 是可调参数。通过选择合适的参数, 比如 $k = 2, n = 5, \alpha = e^{-4}, \beta = 0.75$, AlexNet 利用局部响应归一化的技巧, 可以将在 ImageNet 上的 top-1 与 top-5 错误率分别减少 1.4% 和 1.2%。

4. 重叠池化

池化是一种对窗口数据进行计算的过程, 主要包括平均池化和最大池化。传统的池化窗口是没有重叠的, 不同窗口的池化过程分别独立计算。AlexNet 使用了重叠池化 (overlap pooling), 也就是说, 允许池化窗口重叠。与不重叠池化相比, 重叠池化有助于缓解过拟合, 使 AlexNet 的 top-1 和 top-5 错误率分别降低 0.4% 和 0.3%。

5. 减少过拟合

AlexNet 约有 6000 万个参数, 远远多于 LeNet 的参数。为了减少过拟合, AlexNet 还使

用了数据扩增和丢失输出的训练技巧。数据扩增的方法有两种：一是图像的平移和翻转，二是基于 PCA 的 RGB 强度调整。数据扩增可使 AlexNet 的 top-1 误差率至少减少 1.0%。丢失输出是指在神经网络的训练过程中随机让网络的某些节点（包括输入节点和隐含节点）不工作。使用丢失输出技巧，AlexNet 以 0.5 的概率将两个全连接层神经元的输出设置为零，虽然增加了近一倍收敛所需的迭代次数，但有效阻止了过拟合。

4.2 AlexNet 的 Caffe 代码实现及说明

作为一个具有突破意义的卷积网络模型，AlexNet 已经被集成到 Caffe 框架中，位于 model 子目录下。AlexNet 的 Caffe 实现包括三个文件，分别是网络结构文件 train_val.prototxt、求解器配置文件 solver.prototxt 和伪概率计算文件 deploy.prototxt。其中，train_val.prototxt 用来定义网络的训练数据目录、测试数据目录和网络结构细节，包括每个卷积层的卷积核个数、大小、步长，每个下采样层的类型、窗口大小、移动步长，全连接层的节点数目、激活函数的类型、损失函数类型，以及层与层之间的关系等。solver.prototxt 用来给求解器配置训练和测试网络的有关超参数，包括学习率的大小、训练的迭代次数及优化方法、使用的计算模式（CPU 或 GPU）等。deploy.prototxt 用来在模型训练好后对未知样本计算分类伪概率。下面分别对这三个文件的具体内容进行描述。

1. train_val.prototxt 的代码及说明

```
name: "AlexNet"
layer {
    name: "data"
    type: "Data"
    top: "data"
    top: "label"
    include {
        phase: TRAIN
    }
    transform_param {
        mirror: true      # 镜像翻转
        crop_size: 227    # 裁剪图像块的大小（此处与图 4.1 所示的结构说明不同）
        mean_file: "data/ilsvrc12/imagenet_mean.binaryproto" # 训练集均值的存放位置
    }
    data_param {
        source: "examples/imagenet/ilsvrc12_train_leveldb" # 训练集的存放位置
        batch_size: 256 # 训练集迷你块的大小
        backend: LEVELDB # 一种 Caffe 的数据类型
    }
}
layer {
    name: "data"
    type: "Data"
    top: "data"
    top: "label"
}
```

```

include {
    phase: TEST
}
transform_param {
    mirror: false # 不镜像翻转
    crop_size: 227
    mean_file: "data/ilsrvrc12/imagenet_mean.binaryproto" # 训练集均值的存放位置
}
data_param {
    source: "examples/imagenet/ilsrvrc12_val_leveladb" # 测试集的存放位置
    batch_size: 50 # 测试集迷你块的大小
    backend: LEVELDB # 一种 Caffe 的数据类型
}
}
layer {
    name: "conv1"
    type: "Convolution" # 卷积层
    bottom: "data"
    top: "conv1"
    param {
        lr_mult: 1 # 权值的学习率
        decay_mult: 1 # 权值的衰减率
    }
    param {
        lr_mult: 2 # 偏置的学习率
        decay_mult: 0 # 偏置的衰减率
    }
    convolution_param {
        num_output: 96 # 卷积面的个数
        kernel_size: 11 # 卷积核的大小
        stride: 4 # 卷积核的移动步长
        weight_filler {
            type: "gaussian" # 权值的初始化方法
            std: 0.01 # 权值的初始化标准差
        }
        bias_filler {
            type: "constant" # 把偏置初始化为常数 0
            value: 0
        }
    }
}
}
layer {
    name: "ReLU1"
    type: "ReLU" # ReLU 激活函数层
    bottom: "conv1"
    top: "conv1"
}
}
layer {
    name: "norm1"
    type: "LRN" # 局部响应归一化层
    bottom: "conv1"
    top: "norm1"
    lrn_param {

```

```

        local_size: 5
        alpha: 0.0001
        beta: 0.75
    }
}
layer {
    name: "pool1"
    type: "Pooling"          # 池化层
    bottom: "norm1"
    top: "pool1"
    pooling_param {
        pool: MAX            # 池化类型：最大池化
        kernel_size: 3       # 池化窗口的长和宽
        stride: 2            # 池化窗口的移动步长
    }
}
layer {
    name: "conv2"
    type: "Convolution"      # 卷积层
    bottom: "pool1"
    top: "conv2"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 256      # 卷积面的个数
        pad: 2               # 填充宽度，对该层的输入沿各个方向扩充2个像素宽度，并用0填充
        kernel_size: 5
        group: 2             # 把卷积面分成数目相等的2组
        weight_filler {
            type: "gaussian"
            std: 0.01
        }
        bias_filler {
            type: "constant"
            value: 0.1
        }
    }
}
layer {
    name: "ReLU2"
    type: "ReLU"            # ReLU 激活函数层
    bottom: "conv2"
    top: "conv2"
}
layer {
    name: "norm2"
    type: "LRN"             # 局部响应归一化层

```



```

    bottom: "conv2"
    top: "norm2"
    lrn_param {
        local_size: 5
        alpha: 0.0001
        beta: 0.75
    }
}
layer {
    name: "pool2"
    type: "Pooling"      # 池化层
    bottom: "norm2"
    top: "pool2"
    pooling_param {
        pool: MAX
        kernel_size: 3
        stride: 2
    }
}
layer {
    name: "conv3"
    type: "Convolution"  # 卷积层
    bottom: "pool2"
    top: "conv3"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 384 # 卷积面的个数
        pad: 1          # 填充宽度, 对该层的输入沿各个方向扩充1个像素宽度, 并用0填充
        kernel_size: 3
        weight_filler {
            type: "gaussian"
            std: 0.01
        }
        bias_filler {
            type: "constant"
            value: 0
        }
    }
}
layer {
    name: "ReLU3"
    type: "ReLU"        # ReLU 激活函数层
    bottom: "conv3"
    top: "conv3"
}
layer {

```

```

name: "conv4"
type: "Convolution" # 卷积层
bottom: "conv3"
top: "conv4"
param {
    lr_mult: 1
    decay_mult: 1
}
param {
    lr_mult: 2
    decay_mult: 0
}
convolution_param {
    num_output: 384 # 卷积面的个数
    pad: 1 # 填充宽度, 对该层的输入沿各个方向扩充 1 个像素宽度, 并用 0 填充
    kernel_size: 3
    group: 2
    weight_filler {
        type: "gaussian"
        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0.1
    }
}
}
layer {
    name: "ReLU4"
    type: "ReLU" # ReLU 激活函数层
    bottom: "conv4"
    top: "conv4"
}
layer {
    name: "conv5"
    type: "Convolution" # 卷积层
    bottom: "conv4"
    top: "conv5"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 256 # 卷积面的个数
        pad: 1
        kernel_size: 3
        group: 2
        weight_filler {
            type: "gaussian"

```

```

        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0.1
    }
}
}
layer {
    name: "ReLU5"
    type: "ReLU"           # ReLU 激活函数层
    bottom: "conv5"
    top: "conv5"
}
layer {
    name: "pool5"
    type: "Pooling"        # 池化层
    bottom: "conv5"
    top: "pool5"
    pooling_param {
        pool: MAX          # 池化类型: 最大池化
        kernel_size: 3
        stride: 2
    }
}
}
layer {
    name: "fc6"
    type: "InnerProduct" # 全连接层
    bottom: "pool5"
    top: "fc6"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    inner_product_param {
        num_output: 4096 # 全连接层的节点数
        weight_filler {
            type: "gaussian"
            std: 0.005
        }
        bias_filler {
            type: "constant"
            value: 0.1
        }
    }
}
}
layer {
    name: "ReLU6"
    type: "ReLU"           # ReLU 激活函数层

```



```

        bottom: "fc6"
        top: "fc6"
    }
    layer {
        name: "drop6"
        type: "Dropout"                # 丢失输出层
        bottom: "fc6"
        top: "fc6"
        dropout_param {
            dropout_ratio: 0.5          # 丢失输出的概率
        }
    }
    layer {
        name: "fc7"
        type: "InnerProduct"           # 全连接层
        bottom: "fc6"
        top: "fc7"
        param {
            lr_mult: 1
            decay_mult: 1
        }
        param {
            lr_mult: 2
            decay_mult: 0
        }
        inner_product_param {
            num_output: 4096           # 全连接层的节点数
            weight_filler {
                type: "gaussian"
                std: 0.005
            }
            bias_filler {
                type: "constant"
                value: 0.1
            }
        }
    }
    layer {
        name: "ReLU7"
        type: "ReLU"                   # ReLU 激活函数层
        bottom: "fc7"
        top: "fc7"
    }
    layer {
        name: "drop7"
        type: "Dropout"                # 丢失输出层
        bottom: "fc7"
        top: "fc7"
        dropout_param {
            dropout_ratio: 0.5          # 丢失输出的概率
        }
    }
    layer {

```

```

name: "fc8"
type: "InnerProduct"          # 全连接层
bottom: "fc7"
top: "fc8"
param {
  lr_mult: 1
  decay_mult: 1
}
param {
  lr_mult: 2
  decay_mult: 0
}
inner_product_param {
  num_output: 1000            # 全连接层的节点数
  weight_filler {
    type: "gaussian"
    std: 0.01
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
}
layer {
  name: "accuracy"
  type: "Accuracy"            # 该层只出现在测试阶段，用于计算正确率
  bottom: "fc8"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"      # 带损失的软最大输出层，采用退化交叉熵损失函数
  bottom: "fc8"
  bottom: "label"
  top: "loss"
}

```

2. solver.prototxt 的代码及说明

```

net: "F:/caffe-windows/models/bvlc_AlexNet/train_val.prototxt" # 网络结构文件所在位置
test_iter: 1000          # 测试迭代次数
test_interval: 1000      # 训练 1000 次，测试一次
base_lr: 0.01            # 初始学习率
lr_policy: "step"        # 把学习率的下降策略选为 step 类型，还可以选为 inv、
                          # fixed 和 exp 等类型
gamma: 0.1               # 学习率的策略超参数
stepsizes: 100000        # 每迭代 100 000 次减少一次学习率

```

```

display: 1000           # 每迭代 1000 次，显示 1 次
max_iter: 450000        # 最大迭代次数
momentum: 0.9           # 动量项的加权系数
weight_decay: 0.0005    # 权值衰减系数
snapshot: 10000         # 每训练 10 000 次保存 1 次结果
snapshot_prefix: "F:/caffe-windows/models/bvlc_AlexNet/caffe_AlexNet_train"
solver_mode: GPU

```

3. deploy.prototxt 的代码及说明

```

name: "AlexNet"
layer {
  name: "data"
  type: "Input"
  top: "data"
  input_param { shape: { dim: 10 dim: 3 dim: 227 dim: 227 } } # 输入图像的大小
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 96 # 卷积面的个数
    kernel_size: 11 # 卷积核的大小
    stride: 4 # 卷积核的移动步长
  }
}
layer {
  name: "ReLU1"
  type: "ReLU" # ReLU 激活函数层
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "norm1"
  type: "LRN" # 局部响应归一化层
  bottom: "conv1"
  top: "norm1"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
}

```



```

layer {
  name: "pool1"
  type: "Pooling"
  bottom: "norm1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 256
    pad: 2
    kernel_size: 5
    group: 2
  }
}
layer {
  name: "ReLU2"
  type: "ReLU"          # ReLU 激活函数层
  bottom: "conv2"
  top: "conv2"
}
layer {
  name: "norm2"
  type: "LRN"          # 局部响应归一化
  bottom: "conv2"
  top: "norm2"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
}
layer {
  name: "pool2"
  type: "Pooling"
  bottom: "norm2"
  top: "pool2"
  pooling_param {

```

```

        pool: MAX
        kernel_size: 3
        stride: 2
    }
}
layer {
    name: "conv3"
    type: "Convolution"
    bottom: "pool2"
    top: "conv3"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 384
        pad: 1
        kernel_size: 3
    }
}
layer {
    name: "ReLU3"
    type: "ReLU"           # ReLU 激活函数层
    bottom: "conv3"
    top: "conv3"
}
layer {
    name: "conv4"
    type: "Convolution"
    bottom: "conv3"
    top: "conv4"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 384
        pad: 1
        kernel_size: 3
        group: 2
    }
}
layer {
    name: "ReLU4"
    type: "ReLU"           # ReLU 激活函数层

```

```
        bottom: "conv4"
        top: "conv4"
    }
    layer {
        name: "conv5"
        type: "Convolution"
        bottom: "conv4"
        top: "conv5"
        param {
            lr_mult: 1
            decay_mult: 1
        }
        param {
            lr_mult: 2
            decay_mult: 0
        }
        convolution_param {
            num_output: 256
            pad: 1
            kernel_size: 3
            group: 2
        }
    }
}
layer {
    name: "ReLU5"
    type: "ReLU" # ReLU 激活函数层
    bottom: "conv5"
    top: "conv5"
}
layer {
    name: "pool5"
    type: "Pooling"
    bottom: "conv5"
    top: "pool5"
    pooling_param {
        pool: MAX
        kernel_size: 3
        stride: 2
    }
}
layer {
    name: "fc6"
    type: "InnerProduct" # 全连接层
    bottom: "pool5"
    top: "fc6"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
}
```



```

        inner_product_param {
            num_output: 4096 # 全连接层的节点数
        }
    }
    layer {
        name: "ReLU6"
        type: "ReLU" # ReLU 激活函数层
        bottom: "fc6"
        top: "fc6"
    }
    layer {
        name: "drop6"
        type: "Dropout" # 丢失输出层
        bottom: "fc6"
        top: "fc6"
        dropout_param {
            dropout_ratio: 0.5
        }
    }
    layer {
        name: "fc7"
        type: "InnerProduct" # 全连接层
        bottom: "fc6"
        top: "fc7"
        param {
            lr_mult: 1
            decay_mult: 1
        }
        param {
            lr_mult: 2
            decay_mult: 0
        }
        inner_product_param {
            num_output: 4096 # 全连接层的节点数
        }
    }
    layer {
        name: "ReLU7"
        type: "ReLU" # ReLU 激活函数层
        bottom: "fc7"
        top: "fc7"
    }
    layer {
        name: "drop7"
        type: "Dropout" # 丢失输出层
        bottom: "fc7"
        top: "fc7"
        dropout_param {
            dropout_ratio: 0.5
        }
    }
    layer {
        name: "fc8"
        type: "InnerProduct" # 全连接层
    }

```

```

bottom: "fc7"
top: "fc8"
param {
  lr_mult: 1
  decay_mult: 1
}
param {
  lr_mult: 2
  decay_mult: 0
}
inner_product_param {
  num_output: 1000 # 输出节点的个数
}
}
layer {
  name: "prob"
  type: "Softmax"      # 软最大输出层
  bottom: "fc8"
  top: "prob"
}

```

4.3 AlexNet 的 Caffe 大规模图像分类案例及演示效果

本节描述一个利用 AlexNet 在 Caffe 框架下进行大规模图像分类的案例，其中用到的 ImageNet 数据集可以根据表 1.2 提供的地址下载。

为了在 Caffe 框架下使用 ImageNet 数据集，需要先参考 3.6.1 节将 JPEG 格式的图像转换成 LEVELDB 格式，具体操作命令如方框 4.1 和方框 4.2 所示。

方框 4.1 文件 trainleveldb.bat 的内容

```

SET GLOG_logtostderr=1
F:\caffe-windows\Build\x64\Release\convert_imageset.exe --shuffle=true
--backend=leveldb --resize_height=256 --resize_width=256 F:\train F:\train.
txt G:\dataset\ilsvcr12\ train_leveldb
pause

```

方框 4.2 文件 testleveldb.bat 的内容

```

SET GLOG_logtostderr=1
F:\caffe-windows\Build\x64\Release\convert_imageset.exe --shuffle=true
--backend=leveldb --resize_height=256 --resize_width=256 F:\test F:\test.txt
G:\dataset\ilsvcr12\ test_leveldb
pause

```

另外，AlexNet 在训练和测试过程中还要用到训练集的均值，可以单独调用 compute_image_mean.exe 来计算，也可以执行方框 4.3 所示的批处理文件 meanbinary.bat 来完成。

在 meanbinary.bat 文件中，“--backend=leveldb”表示把数据转换为 LEVELDB 格式；“F:\train.txt”表示训练集的标签文件；“G:\dataset\ilsvcr12\train_leveldb”表示转换后训练集所在目录；“G:\dataset\ilsvcr12\imagenet_mean.binaryproto”表示转换后训练集均值的存

储文件。

方框 4.3 文件 meanbinary.bat 的内容

```
SET GLOG_logtostderr=1
F:\caffe-windows\Build\x64\Release\compute_image_mean.exe --backend=leveldb
F:\train.txt G:\dataset\ilsvcr12\train_leveldb G:\dataset\ilsvcr12\imagenet_
mean.binaryproto
pause
```

在建立好 ImageNet 的训练集、测试集和均值后，利用 AlexNet 对 ImageNet 数据集分类，只需根据 train_val.prototxt 和 solver.prototxt 文件中设置的参数值，参照图 4.2 的命令运行程序。程序运行后，将依次出现一系列信息窗口界面，如图 4.3 和图 4.4 所示。图 4.3 显示的网络结构信息其实是对文件 train_val.prototxt 的加载过程。图 4.4 显示的是在训练网络的过程中，随着迭代次数的增加，学习率、损失函数、训练正确率和测试正确率的变化。训练完成时的最终结果如图 4.5 所示，迭代训练 450 000 次后，程序运行结束，训练损失函数值为 0.570 359，测试损失函数值为 1.835 93，测试准确率约为 57.04%。

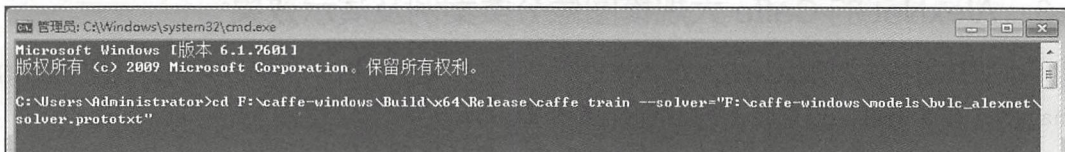


图 4.2 AlexNet 大规模图像分类案例的 Caffe 运行命令

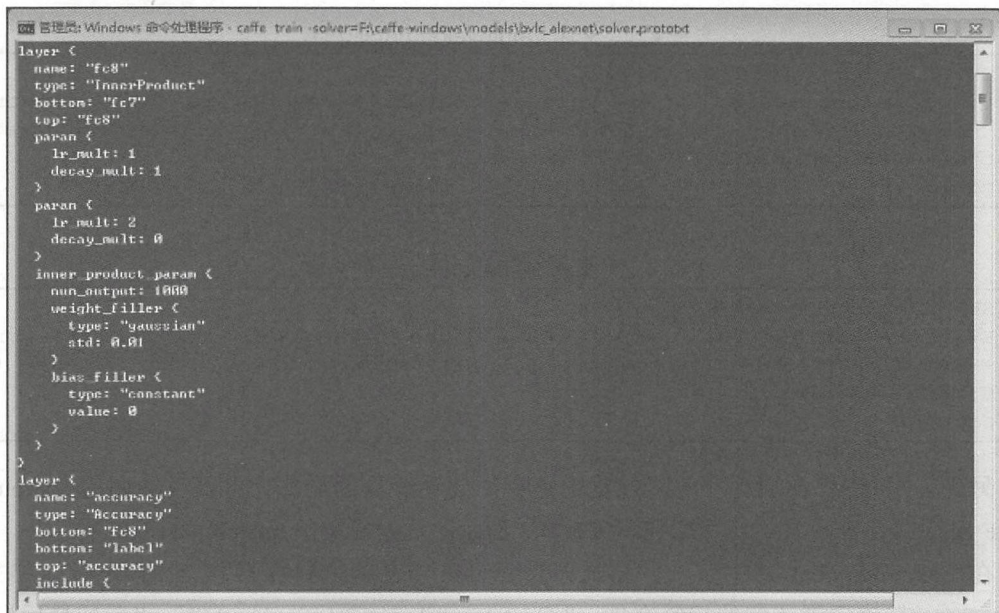


图 4.3 AlexNet 大规模图像分类案例运行后 Caffe 显示的网络结构信息


```

管理员: Windows 命令处理程序 - caffe train -solver=F:\caffe-windows\models\bvlc_alexnet\solver.prototxt
10602 12:56:19.419409 42808 solver.cpp:2581 Train net output #0: accuracy = 0.160156
10602 12:56:19.419409 42808 solver.cpp:2581 Train net output #1: loss = 4.23754 (* 1 = 4.23754 loss)
10602 12:56:19.420409 42808 sgd_solver.cpp:1061 Iteration 5000, lr = 0.01
10602 13:22:16.280457 42808 solver.cpp:3461 Iteration 6000, Testing net (#0)
10602 13:24:13.001132 42808 solver.cpp:4141 Test net output #0: accuracy = 0.18994
10602 13:24:13.001132 42808 solver.cpp:4141 Test net output #1: loss = 4.09525 (* 1 = 4.09525 loss)
10602 13:24:14.308207 42808 solver.cpp:2421 Iteration 6000, loss = 4.47051
10602 13:24:14.308207 42808 solver.cpp:2581 Train net output #0: accuracy = 0.140625
10602 13:24:14.308207 42808 solver.cpp:2581 Train net output #1: loss = 4.47051 (* 1 = 4.47051 loss)
10602 13:24:14.308207 42808 sgd_solver.cpp:1061 Iteration 6000, lr = 0.01
10602 13:50:11.233258 42808 solver.cpp:3461 Iteration 7000, Testing net (#0)
10602 13:52:08.005937 42808 solver.cpp:4141 Test net output #0: accuracy = 0.20972
10602 13:52:08.006937 42808 solver.cpp:4141 Test net output #1: loss = 3.97034 (* 1 = 3.97034 loss)
10602 13:52:09.312012 42808 solver.cpp:2421 Iteration 7000, loss = 4.11723
10602 13:52:09.312012 42808 solver.cpp:2581 Train net output #0: accuracy = 0.164063
10602 13:52:09.312012 42808 solver.cpp:2581 Train net output #1: loss = 4.11723 (* 1 = 4.11723 loss)
10602 13:52:09.313012 42808 sgd_solver.cpp:1061 Iteration 7000, lr = 0.01
10602 14:18:06.310067 42808 solver.cpp:3461 Iteration 8000, Testing net (#0)
10602 14:20:03.078747 42808 solver.cpp:4141 Test net output #0: accuracy = 0.21336
10602 14:20:03.078747 42808 solver.cpp:4141 Test net output #1: loss = 3.94317 (* 1 = 3.94317 loss)
10602 14:20:04.386821 42808 solver.cpp:2421 Iteration 8000, loss = 4.12209
10602 14:20:04.386821 42808 solver.cpp:2581 Train net output #0: accuracy = 0.203125
10602 14:20:04.386821 42808 solver.cpp:2581 Train net output #1: loss = 4.12209 (* 1 = 4.12209 loss)
10602 14:20:04.386821 42808 sgd_solver.cpp:1061 Iteration 8000, lr = 0.01
10602 14:46:01.180865 42808 solver.cpp:3461 Iteration 9000, Testing net (#0)
10602 14:47:57.950543 42808 solver.cpp:4141 Test net output #0: accuracy = 0.23524
10602 14:47:57.950543 42808 solver.cpp:4141 Test net output #1: loss = 3.77094 (* 1 = 3.77094 loss)
10602 14:47:59.256618 42808 solver.cpp:2421 Iteration 9000, loss = 3.90535
10602 14:47:59.256618 42808 solver.cpp:2581 Train net output #0: accuracy = 0.21875
10602 14:47:59.256618 42808 solver.cpp:2581 Train net output #1: loss = 3.90535 (* 1 = 3.90535 loss)
10602 14:47:59.257618 42808 sgd_solver.cpp:1061 Iteration 9000, lr = 0.01

```

图 4.4 AlexNet 大规模图像分类案例运行后 Caffe 显示的训练和测试信息

```

C:\Windows\system32\cmd.exe
.11326 loss>
10618 01:46:40.665868 8040 sgd_solver.cpp:1061 Iteration 448000, lr = 1e-006
10618 02:02:08.212296 8040 solver.cpp:3371 Iteration 449000, Testing net (#0)
10618 02:02:08.212296 8040 net.cpp:6841 Ignoring source layer accuracy1
10618 02:03:06.088398 8040 solver.cpp:4041 Test net output #0: accuracy = 0.57042
10618 02:03:06.088398 8040 solver.cpp:4041 Test net output #1: loss = 1.83547 (* 1 = 1.83547 loss)
10618 02:03:06.353600 8040 solver.cpp:2281 Iteration 449000, loss = 1.13231
10618 02:03:06.353600 8040 solver.cpp:2441 Train net output #0: accuracy = 0.703125
10618 02:03:06.353600 8040 solver.cpp:2441 Train net output #1: loss = 1.13231 (* 1 = 1.13231 loss)
10618 02:03:06.353600 8040 sgd_solver.cpp:1061 Iteration 449000, lr = 1e-006
10618 02:18:34.087229 8040 solver.cpp:4541 Snapshotting to binary proto file D:\caffe-windows\models\bvlc_alexnet\caffe_alexnet_train_iter_450000.caffemodel
10618 02:18:36.068433 8040 sgd_solver.cpp:2731 Snapshotting solver state to binary proto file D:\caffe-windows\models\bvlc_alexnet\caffe_alexnet_train_iter_450000.solverstate
10618 02:18:36.723634 8040 solver.cpp:3171 Iteration 450000, loss = 1.21678
10618 02:18:36.723634 8040 solver.cpp:3371 Iteration 450000, Testing net (#0)
10618 02:18:36.723634 8040 net.cpp:6841 Ignoring source layer accuracy1
10618 02:19:34.038133 8040 solver.cpp:4041 Test net output #0: accuracy = 0.570359
10618 02:19:34.038133 8040 solver.cpp:4041 Test net output #1: loss = 1.83593 (* 1 = 1.83593 loss)
10618 02:19:34.038133 8040 solver.cpp:3221 Optimization Done.
10618 02:19:34.038133 8040 caffe.cpp:2231 Optimization Done.

D:\caffe-windows\models\bvlc_alexnet>pause
请按任意键继续. . .

```

图 4.5 AlexNet 大规模图像分类案例训练完成时 Caffe 的最终结果

4.4 AlexNet 的 TensorFlow 代码实现及说明

利用 TensorFlow 实现 AlexNet 主要有三个程序，分别是 input_data.py、model.py 和 training.py。其中，input_data.py 用来定义网络的输入情况，包括网络训练数据和测试数据

转换为 TensorFlow 能识别的数据队列形式，以及重置大小和归一化等。model.py 用来定义训练和测试时网络的结构情况，包括卷积层和下采样层的个数及排列形式、卷积层中卷积核的个数和大小、下采样层中滑动的窗口的大小和滑动窗口的移动步长、全连接层的节点个数、激活函数的类型，以及损失函数、训练学习函数和准确率测试函数等。training.py 用来定义网络的训练情况，包括文件的输入和存储路径、训练时的各个参数大小、训练过程中训练情况的打印显示等。下面分别对这三个文件的具体内容进行描述。

1. input_data.py 的代码及说明

```
import tensorflow as tf
import numpy as np
import os

def get_files(file_dir, label_file):
    # 对原始图像及标签的数据格式进行转换
    image_list = []
    label_list = []
    len_file = len(os.listdir(file_dir)) # 统计指定文件夹中的图像文件个数
    len_label = len(open(label_file).readlines()) # 统计 label_file 文件有多少行
    if len_file == len_label: # 判断图像文件个数和标签个数是否相等
        print('num of images identify to num of labels.')
        print('The len of file is %d.' % len_file)
    txt_file = open(label_file, 'r')
    for file in os.listdir(file_dir):
        image_list.append(file_dir + file)
        one_content = txt_file.readline()
        name = one_content.split(sep=' ')
        if name[0] == file:
            name1 = name[1].split(sep='\n')
            label_list.append(int(name1[0]) - 1)
        else:
            print('File name is different from label name!\n') # 错误时打印此信息
    txt_file.close()
    print('There are %d images\nThere are %d labels \n' % (len(image_list), len(label_list)))
    temp = np.array([image_list, label_list]) # 把图像名序列和标签序列做成一个二维数组
    temp = temp.transpose() # 对二维数组转置
    np.random.shuffle(temp) # 打乱数组各行顺序
    image_list = list(temp[:, 0])
    label_list = list(temp[:, 1])
    label_list = [int(i) for i in label_list] # 把字符型标签转换成整数型标签
    return image_list, label_list

def get_batch(image_list, label_list, image_W, image_H, batch_size, capacity):
    # 对数据进行分块
    image_cast = tf.cast(image_list, tf.string) # 将 python.list 类型转换成 tf 能够识别的格式
    label_cast = tf.cast(label_list, tf.int32)
    input_queue = tf.train.slice_input_producer([image_cast, label_cast]) # 生成队列
    labels = input_queue[1]
```



```

image_contents = tf.read_file(input_queue[0])
images = tf.image.decode_jpeg(image_contents, channels=3)
images = tf.image.resize_image_with_crop_or_pad(images, image_W, image_H)
images = tf.image.per_image_standardization(images) # 标准化数据
image_batch, label_batch = tf.train.batch([images, labels], batch_size= batch_size, num_threads= 64, capacity = capacity)
label_batch = tf.reshape(label_batch, [batch_size])
image_batch = tf.cast(image_batch, tf.float32)
return image_batch, label_batch

```

2. model.py 的代码及说明

```

import TensorFlow as tf
def inference(images, batch_size, n_classes): # 定义 AlexNet 网络的结构
    with tf.variable_scope('conv1_lrn') as scope: # 卷积层 conv1
        weights = tf.get_variable('weights', shape = [11,11,3, 96], dtype = tf.float32,
            initializer=tf.truncated_normal_initializer(stddev=0.01,dtype=tf.float32))
        # 定义权值并初始化
        biases = tf.get_variable('biases', shape=[96], dtype=tf.float32, initializer=\
            tf.constant_initializer(0.1)) # 定义偏置并初始化
        conv = tf.nn.conv2d(images, weights, strides=[1,4,4,1], padding='SAME')
        # 卷积运算
        pre_activation = tf.nn.bias_add(conv, biases)
        conv1 = tf.nn.ReLU(pre_activation, name= scope.name) # 使用 ReLU 激活函数
        norm1 = tf.nn.lrn(conv1, depth_radius=4, bias=1.0, alpha=0.001/9.0,
            beta=0.75,name='norm1') # conv1 的局部响应归一化
    with tf.variable_scope('pooling1') as scope: # 池化层 pool1
        pool1 = tf.nn.max_pool(norm1, ksize=[1,3,3,1],strides=[1,2,2,1],
            padding='VALID', name='pooling1') # 使用最大池化
    with tf.variable_scope('conv2') as scope: # 卷积层 conv2
        weights = tf.get_variable('weights', # 定义权值并初始化
            shape=[5,5,96,256], dtype=tf.float32, initializer=tf.truncated_\
            normal_initializer(stddev=0.01,dtype=tf.float32))
        biases = tf.get_variable('biases', shape=[256], dtype=tf.float32, initializer=\
            tf.constant_initializer(0.1)) # 定义偏置并初始化
        conv = tf.nn.conv2d(pool1, weights, strides=[1,1,1,1],padding='SAME')
        # 卷积运算
        pre_activation = tf.nn.bias_add(conv, biases)
        conv2 = tf.nn.ReLU(pre_activation, name='conv2') # 使用 ReLU 激活函数
    with tf.variable_scope('pooling2_lrn') as scope: # 池化层 pool2
        norm2 = tf.nn.lrn(conv2, depth_radius=4, bias=1.0, alpha=0.001/9.0,
            beta=0.75,name='norm2') # 对 conv2 的局部响应归一化
        pool2 = tf.nn.max_pool(norm2, ksize=[1,3,3,1], strides=[1,2,2,1],
            padding='VALID',name='pooling2') # 使用最大池化
    with tf.variable_scope('conv3') as scope: # 卷积层 conv3
        weights = tf.get_variable('weights', shape=[3,3,256,384], dtype=tf.float32,
            initializer=tf.truncated_normal_initializer(stddev=0.01,dtype=tf.float32))
        # 定义权值并初始化
        biases = tf.get_variable('biases', shape=[384], dtype=tf.float32, initializer=\
            tf.constant_initializer(0.1)) # 定义偏置并初始化

```



```

conv = tf.nn.conv2d(pool2, weights, strides=[1,1,1,1],padding='SAME')
pre_activation = tf.nn.bias_add(conv, biases)
conv3 = tf.nn.ReLU(pre_activation, name='conv3')
with tf.variable_scope('conv4') as scope: # 卷积层 conv4
    weights = tf.get_variable('weights', shape=[3,3,384,384], dtype=tf.float32,
        initializer=tf.truncated_normal_initializer(stddev=0.01,dtype=tf.float32))
    # 定义权值并初始化
    biases = tf.get_variable('biases', shape=[384], dtype=tf.float32, initializer=\
        tf.constant_initializer(0.1)) # 定义偏置并初始化
    conv = tf.nn.conv2d(conv3, weights, strides=[1,1,1,1],padding='SAME')
    pre_activation = tf.nn.bias_add(conv, biases)
    conv4 = tf.nn.ReLU(pre_activation, name='conv4')
with tf.variable_scope('conv5') as scope: # 卷积层 conv5
    weights = tf.get_variable('weights', shape=[3,3,384,256], dtype=tf.float32,
        initializer=tf.truncated_normal_initializer(stddev=0.1,dtype=tf.float32))
    # 定义权值并初始化
    biases = tf.get_variable('biases', shape=[256], dtype=tf.float32, initializer=\
        tf.constant_initializer(0.1)) # 定义偏置并初始化
    conv = tf.nn.conv2d(conv4, weights, strides=[1,1,1,1], padding='SAME')
    pre_activation = tf.nn.bias_add(conv, biases)
    conv5 = tf.nn.ReLU(pre_activation, name='conv5')
with tf.variable_scope('pooling6') as scope: # 池化层 pool6
    pool6 = tf.nn.max_pool(conv5, ksize=[1,3,3,1], strides=[1,2,2,1], padding=\
        'VALID',name='pooling6') # 使用最大池化
with tf.variable_scope('local7') as scope: # 全连接层 local7
    reshape = tf.reshape(pool2, shape=[batch_size, -1])
    dim = reshape.get_shape()[1].value
    weights = tf.get_variable('weights', shape=[dim,4096], dtype=tf.float32,
        initializer=tf.truncated_normal_initializer(stddev=0.005,dtype=tf.float32))
    # 定义权值并初始化
    biases = tf.get_variable('biases', shape=[4096], dtype=tf.float32,
        initializer=tf.constant_initializer(0.1)) # 定义偏置并初始化
    local7 = tf.nn.ReLU(tf.matmul(reshape, weights) + biases, name=scope.name)
    local7 = tf.nn.dropout(local7, keep_prob=0.5) # 使用丢失输出技巧
with tf.variable_scope('local8') as scope: # 全连接层 local8
    weights = tf.get_variable('weights', shape=[4096,4096], dtype=tf.float32,
        initializer=tf.truncated_normal_initializer(stddev=0.005,dtype=tf.float32))
    # 定义权值并初始化
    biases = tf.get_variable('biases', shape=[4096], dtype=tf.float32,
        initializer=tf.constant_initializer(0.1)) # 定义偏置并初始化
    local8 = tf.nn.ReLU(tf.matmul(local7, weights) + biases, name='local8')
    local8 = tf.nn.dropout(local8, keep_prob=0.5) # 使用丢失输出技巧
with tf.variable_scope('softmax_linear') as scope: # 线性软最大输出层
    weights = tf.get_variable('softmax_linear', shape=[4096, n_classes],
        dtype=tf.float32, initializer=tf.truncated_normal_initializer(stddev=0.005,
        dtype=tf.float32)) # 定义权值并初始化
    biases = tf.get_variable('biases', shape=[n_classes], dtype=tf.float32,
        initializer=tf.constant_initializer(0.1)) # 定义偏置并初始化
    softmax_linear = tf.add(tf.matmul(local8, weights), biases, name='softmax_\
        linear')

```

```

    return softmax_linear
def losses(logits, labels):
    # 定义损失函数
    with tf.variable_scope('loss') as scope:
        cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits\
            (logits=logits, labels=labels, name='xentropy_per_example')
        loss = tf.reduce_mean(cross_entropy, name='loss') # 计算张量的平均值
        tf.summary.scalar(scope.name+'/loss', loss) # 对 loss 汇总和记录
    return loss
def training(loss, learning_rate):
    # 定义训练函数
    with tf.name_scope('optimizer'):
        optimizer = tf.train.AdamOptimizer(learning_rate= learning_rate)
        # 使用 Adam 优化器
        global_step = tf.Variable(0, name='global_step', trainable=False)
        # 初始化 global_step 为 0
        train_op = optimizer.minimize(loss, global_step= global_step)
        # 更新参数最小化损失
    return train_op
def evaluation(logits, labels):
    # 定义评估函数, 计算准确率
    with tf.variable_scope('accuracy') as scope:
        correct = tf.nn.in_top_k(logits, labels, 1)
        # 判断标记是否在前 k 个预测中, 返回 bool 型 Tensor
        correct = tf.cast(correct, tf.float16) # 将 bool 型数据转换为 float16
        accuracy = tf.reduce_mean(correct) # 通过计算 correct 的均值得到准确率
        tf.summary.scalar(scope.name+'/accuracy', accuracy) # 对准确率数据进行汇总
    return accuracy

```

3. training.py 的代码及说明

```

import os
import numpy as np
import TensorFlow as tf
import input_data
import model
N_CLASSES = 1000 # 输出类别数
IMG_W = 227 # 图像宽度
IMG_H = 227 # 图像高度
TRAIN_BATCH_SIZE = 16 # 训练集的批大小
VAL_BATCH_SIZE = 32 # 测试集的批大小
CAPACITY = 2000 # 用于定义的范围
MAX_STEP = 450000 # 最大的迭代步
learning_rate = 0.0001 # 学习率
def run_training():
    # 定义训练函数
    train_dir = 'C:/Python35/my_imagenet_code/data/train/' # 训练集目录
    train_labelfile = 'C:/Python35/my_imagenet_code/data/train.txt'
    # 训练集标签目录
    logs_train_dir = 'C:/Python35/my_imagenet_code/data/logs/train/'
    # 训练集训练参数存放目录
    val_dir = 'C:/Python35/my_imagenet_code/data/val/' # 测试集目录
    val_labelfile = 'C:/Python35/my_imagenet_code/data/val.txt' # 测试集标签目录
    logs_val_dir = 'C:/Python35/my_imagenet_code/data/logs/val/'
    # 测试集训练参数存放目录
    train_list, trainlabel_list = input_data.get_files(train_dir, train_labelfile)

```

```

# 获取训练集及其标签
val_list, vallabel_list = input_data.get_files(val_dir, val_label_file)
# 获取测试集及其标签
train_batch, train_label_batch = input_data.get_batch(train_list,
                                                         trainlabel_list, IMG_W, IMG_H,
                                                         TRAIN_BATCH_SIZE, CAPACITY)
# 获取训练集的 batch 及其标签
val_batch, val_label_batch = input_data.get_batch(val_list,
                                                    vallabel_list, IMG_W, IMG_H,
                                                    VAL_BATCH_SIZE, CAPACITY)
# 获取训练集的 batch 及其标签
logits = model.inference(train_batch, TRAIN_BATCH_SIZE, N_CLASSES)
# 获取训练 batch 网络输出结果
loss = model.losses(logits, train_label_batch) # 计算训练 batch 的损失
train_op = model.training(loss, learning_rate) # 利用损失和学习率更新参数
acc = model.evaluation(logits, train_label_batch) # 计算准确率

x_train = tf.placeholder(tf.float32, shape=[TRAIN_BATCH_SIZE, IMG_W, IMG_H, 3])
y_train_ = tf.placeholder(tf.int16, shape=[TRAIN_BATCH_SIZE])
# 用于得到传递进来的真实训练样本
x_val = tf.placeholder(tf.float32, shape=[VAL_BATCH_SIZE, IMG_W, IMG_H, 3])
y_val_ = tf.placeholder(tf.int16, shape=[VAL_BATCH_SIZE])
# 用于得到传递进来的真实训练样本
with tf.Session() as sess:
    saver = tf.train.Saver()
    sess.run(tf.global_variables_initializer()) # 运行初始化
    coord = tf.train.Coordinator() # 设置多线程协调器
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)
    # 开始队列运行器 (Queue Runner)
    summary_op = tf.summary.merge_all() # 汇总操作
    train_writer = tf.summary.FileWriter(logs_train_dir, sess.graph)
    # 把训练的汇总写入 logs_train_dir
    val_writer = tf.summary.FileWriter(logs_val_dir, sess.graph)
    # 把测试的汇总写入 logs_val_dir
    try: # 开始运行
        for step in np.arange(MAX_STEP):
            if coord.should_stop():
                break
            tra_images, tra_labels = sess.run([train_batch, train_label_ \
                                                batch])
            _, tra_loss, tra_acc = sess.run([train_op, loss, acc], feed_ \
                                             dict={x_train: tra_images, y_train_: tra_labels})
            # 计算训练损失和准确率
            if step % 50 == 0: # 打印训练集准确率
                print('Step %d, train loss = %.2f, train accuracy = %.2f%%' \
                      % (step, tra_loss, tra_acc*100.0))
                summary_str = sess.run(summary_op)
                train_writer.add_summary(summary_str, step)
            if step % 200 == 0 or (step + 1) == MAX_STEP: # 打印测试集准确率
                val_images, val_labels = sess.run([val_batch, val_label_batch])
                val_loss, val_acc = sess.run([loss, acc], feed_dict={x_val: val_ \
                                                                      images, y_val_: val_labels}) # 计算测试损失和准确率
                print('** Step %d, val loss = %.2f, val accuracy = %.2f%%'

```



```

        ''' %(step, val_loss, val_acc*100.0))
        summary_str = sess.run(summary_op)
        val_writer.add_summary(summary_str, step)
    if step % 2000 == 0 or (step + 1) == MAX_STEP: # 保存模型
        checkpoint_path = os.path.join(logs_train_dir, 'model.ckpt')
        saver.save(sess, checkpoint_path, global_step=step)
except tf.errors.OutOfRangeError:
    print('Done training -- epoch limit reached')
finally:
    coord.request_stop()
    coord.join(threads)
if __name__ == "__main__": # 程序从这里开始运行
    run_training()

```

4.5 AlexNet 的 TensorFlow 大规模图像分类案例及演示效果

本案例需要重新组织 4.3 节的 ImageNet 数据集。首先，将其存放在文件夹 E:\ILSVRC2012 下，其中 120 万幅训练图像存放在子文件夹 train 下（如图 4.6 所示），5 万幅测试图像存放在子文件夹 val 下（如图 4.7 所示）。然后，在文件夹 E:\ILSVRC2012 下创建训练标签文件 train.txt 和测试标签文件 val.txt，其中 train.txt 由训练图像名称加标签构成（如图 4.8 所示），val.txt 由测试图像名称加标签构成（如图 4.9 所示）。

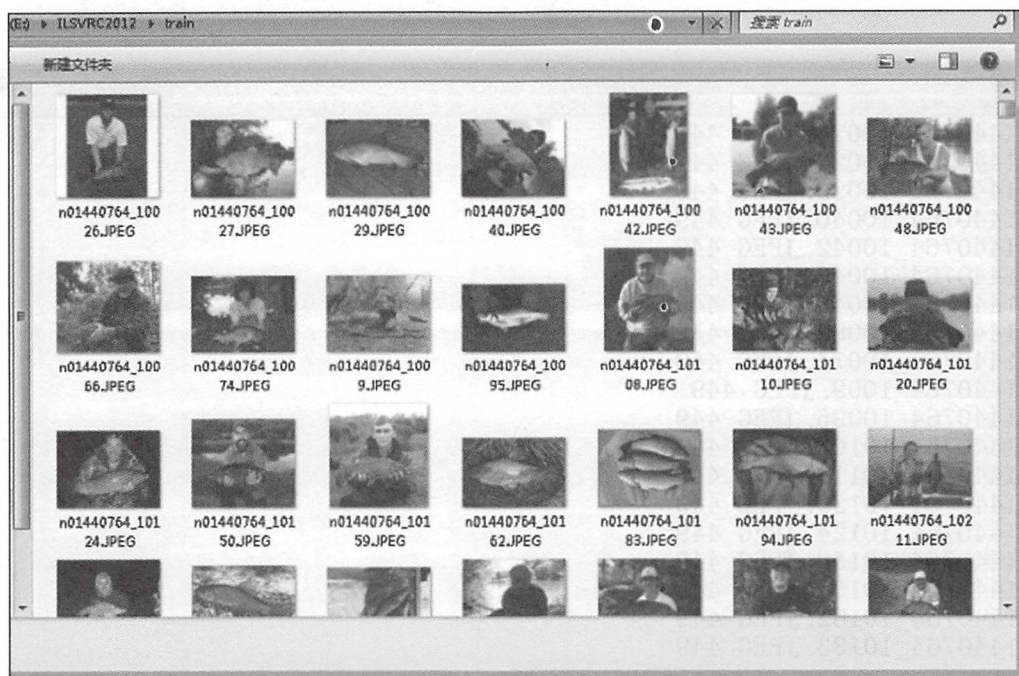


图 4.6 子文件夹 train 下的部分 ImageNet 训练图像样本





图 4.7 子文件夹 val 下的部分 ImageNet 测试图像样本

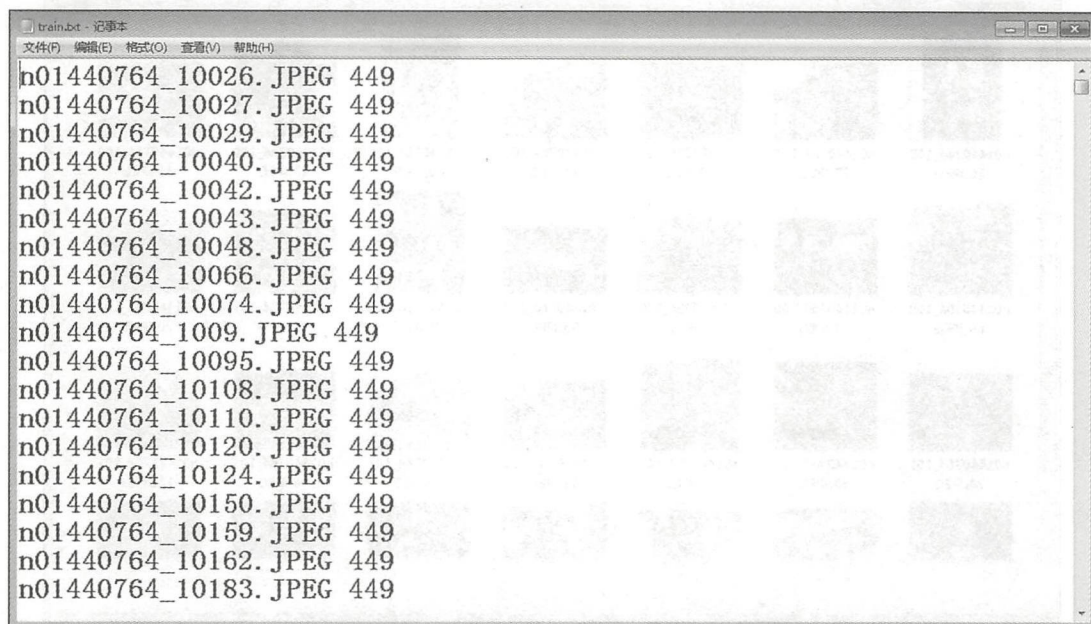


图 4.8 训练标签文件 train.txt 的部分内容



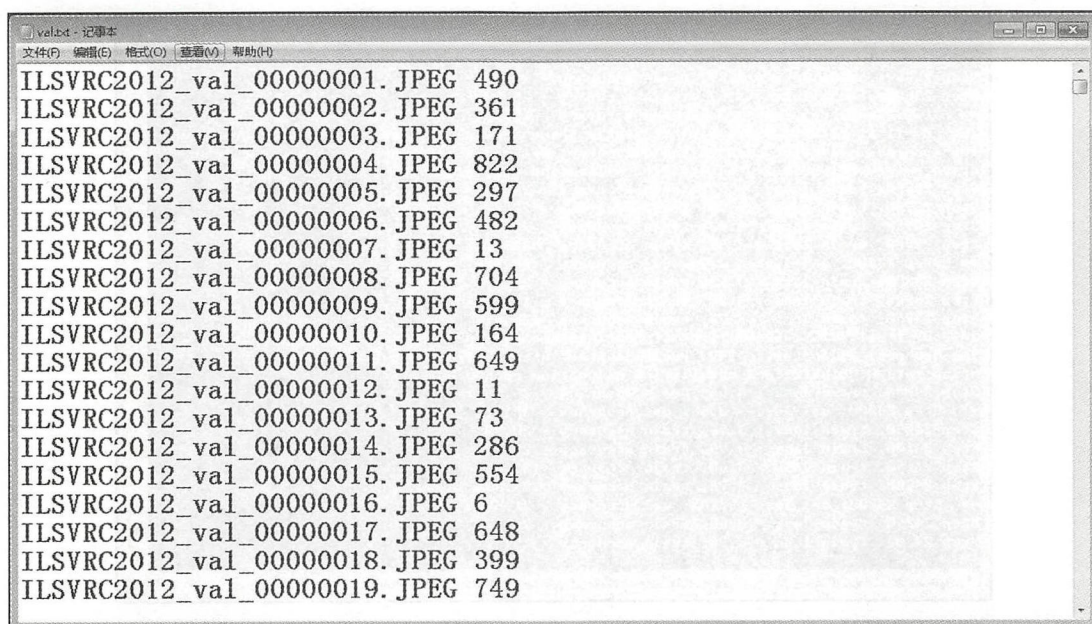


图 4.9 测试标签文件 val.txt 的部分信息

在做好上述准备工作后, 根据 4.4 节程序 `input_data.py`、`model.py` 和 `training.py` 的设置, 参照图 4.10 的命令运行。程序运行后, 将出现一系列信息窗口界面, 如图 4.11 所示。图 4.11 显示的是在训练网络的过程中, 随着迭代次数的增加, 损失函数值、训练准确率和测试准确率的变化。训练完成时的最终结果如图 4.12 所示, 迭代训练 450 000 次后, 程序运行结束, 训练损失函数值为 0.78, 测试损失函数值为 0.98, 验证准确率为 73.83%。注意, 本实验取得的结果明显优于上面 Caffe 的结果, 可能是由于二者训练网络时使用的优化策略不同。Caffe 使用 “SGD”, TensorFlow 使用 “Adam”, 最终导致损失结果不同。如图 4.5 和图 4.12 所示, Caffe 在训练结束时的验证损失为 1.835 93、验证准确率约为 57.04%, TensorFlow 在训练结束时的验证损失为 0.90、验证准确率约为 73.83%。

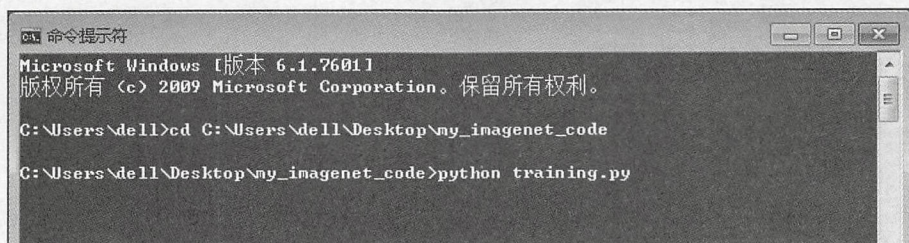


图 4.10 AlexNet 大规模图像分类案例的 TensorFlow 运行命令




```

Ca\Windows\system32\cmd.exe - python training.py
Step 23500, train loss = 3.66, train accuracy = 26.95%
Step 23550, train loss = 3.89, train accuracy = 22.27%
Step 23600, train loss = 3.67, train accuracy = 25.00%
** Step 23600, val loss = 3.69, val accuracy = 19.92% **
2017-07-11 11:22:59.409997: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\wind
ime\gpu\pool_allocator.cc:2471 PoolAllocator: After 379159147 get requests, put_count=379
ction_rate=0.00169058 and unsatisfied allocation rate=0.0016906
Step 23650, train loss = 3.77, train accuracy = 24.61%
Step 23700, train loss = 3.79, train accuracy = 22.27%
Step 23750, train loss = 3.82, train accuracy = 23.83%
Step 23800, train loss = 3.59, train accuracy = 32.81%
** Step 23800, val loss = 3.74, val accuracy = 24.22% **
Step 23850, train loss = 3.53, train accuracy = 29.69%
Step 23900, train loss = 3.90, train accuracy = 17.19%
Step 23950, train loss = 3.72, train accuracy = 24.61%
2017-07-11 11:39:55.352469: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\wind
ime\gpu\pool_allocator.cc:2471 PoolAllocator: After 384930060 get requests, put_count=384
ction_rate=0.00169122 and unsatisfied allocation rate=0.00169122
Step 24000, train loss = 3.65, train accuracy = 26.56%
** Step 24000, val loss = 3.68, val accuracy = 32.81% **
Step 24050, train loss = 3.96, train accuracy = 20.70%
Step 24100, train loss = 3.78, train accuracy = 28.52%
Step 24150, train loss = 3.60, train accuracy = 25.39%
Step 24200, train loss = 3.78, train accuracy = 23.83%
** Step 24200, val loss = 3.88, val accuracy = 21.48% **
Step 24250, train loss = 3.58, train accuracy = 26.17%
Step 24300, train loss = 3.76, train accuracy = 25.39%

```

图 4.11 AlexNet 大规模图像分类案例运行后 TensorFlow 显示的训练和验证信息

```

Ca\Windows\system32\cmd.exe
Step 448850, train loss = 0.75, train accuracy = 80.47%
2017-08-03 10:43:22.060758: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\common_run
ime\gpu\pool_allocator.cc:2471 PoolAllocator: After 7225187978 get requests, put_count=8994000
eviction_rate=0.00124481 and unsatisfied allocation rate=0.00124481
Step 448900, train loss = 0.67, train accuracy = 84.77%
Step 448950, train loss = 0.76, train accuracy = 77.34%
Step 449000, train loss = 0.70, train accuracy = 80.08%
**** Step 449000, val loss = 0.67, val accuracy = 85.16% ****
Step 449050, train loss = 0.64, train accuracy = 82.81%
Step 449100, train loss = 0.68, train accuracy = 81.64%
Step 449150, train loss = 0.71, train accuracy = 78.52%
Step 449200, train loss = 0.85, train accuracy = 76.56%
**** Step 449200, val loss = 0.79, val accuracy = 77.73% ****
Step 449250, train loss = 0.87, train accuracy = 77.73%
Step 449300, train loss = 0.82, train accuracy = 74.61%
2017-08-03 11:05:24.391091: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\common_run
ime\gpu\pool_allocator.cc:2471 PoolAllocator: After 7232595343 get requests, put_count=7232595342 evicted_count=9004000
eviction_rate=0.00124492 and unsatisfied allocation rate=0.00124492
Step 449350, train loss = 0.69, train accuracy = 80.47%
Step 449400, train loss = 0.81, train accuracy = 75.78%
**** Step 449400, val loss = 0.63, val accuracy = 82.03% ****
Step 449450, train loss = 0.75, train accuracy = 78.91%
Step 449500, train loss = 0.79, train accuracy = 76.95%
Step 449550, train loss = 0.82, train accuracy = 76.95%
Step 449600, train loss = 0.74, train accuracy = 80.08%
**** Step 449600, val loss = 0.83, val accuracy = 78.12% ****
Step 449650, train loss = 0.56, train accuracy = 83.59%
Step 449700, train loss = 0.81, train accuracy = 78.52%
Step 449750, train loss = 0.71, train accuracy = 79.30%
Step 449800, train loss = 0.88, train accuracy = 79.69%
**** Step 449800, val loss = 0.80, val accuracy = 78.12% ****
Step 449850, train loss = 0.71, train accuracy = 78.52%
2017-08-03 11:30:28.920134: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\common_run
ime\gpu\pool_allocator.cc:2471 PoolAllocator: After 7240893054 get requests, put_count=7240893054 evicted_count=9014000
eviction_rate=0.00124487 and unsatisfied allocation rate=0.00124487
Step 449900, train loss = 0.83, train accuracy = 78.12%
Step 449950, train loss = 0.78, train accuracy = 77.73%
**** Step 449999, val loss = 0.90, val accuracy = 73.83% ****
C:\Users\dell\Desktop\my_inagenet_code>

```

图 4.12 AlexNet 大规模图像分类案例的 TensorFlow 最终验证结果



4.6 AlexNet 的改进模型 ZFNet

ZFNet 是 AlexNet 的一种改进模型，其结构如图 4.13 所示，各层细节如表 4.3 所示。

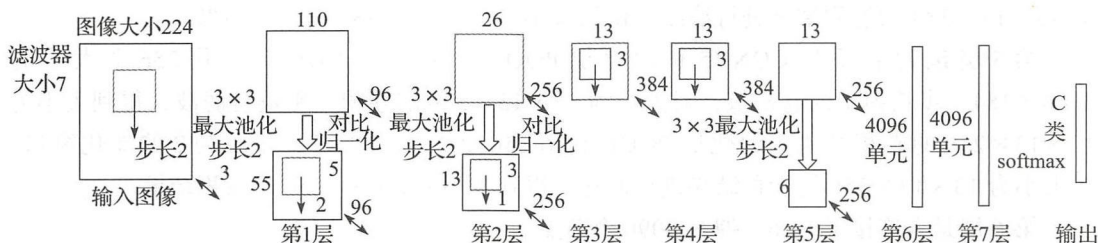


图 4.13 ZFNet 的模型结构示意图

表 4.3 ZFNet 各层的结构细节及参数情况

层	类型	输入	核大小	步长	输出	参数个数
第 1 层	CONV1	$224 \times 224 \times 3$	7×7	2	$110 \times 110 \times 96$	$(7 \times 7 \times 3 + 1) \times 96$
	POOL1	$110 \times 110 \times 96$	3×3	2	$55 \times 55 \times 96$	0
第 2 层	CONV2	$55 \times 55 \times 96$	5×5	2	$26 \times 26 \times 256$	$(5 \times 5 \times 96 + 1) \times 256$
	POOL2	$26 \times 26 \times 256$	3×3	2	$13 \times 13 \times 256$	0
第 3 层	CONV3	$13 \times 13 \times 256$	3×3	1	$13 \times 13 \times 384$	$(3 \times 3 \times 256 + 1) \times 384$
第 4 层	CONV4	$13 \times 13 \times 384$	3×3	1	$13 \times 13 \times 384$	$(3 \times 3 \times 384 + 1) \times 384$
第 5 层	CONV5	$13 \times 13 \times 384$	3×3	1	$13 \times 13 \times 256$	$(3 \times 3 \times 384 + 1) \times 256$
	POOL3	$13 \times 13 \times 256$	3×3	2	$6 \times 6 \times 256$	0
第 6 层	FC6	$1 \times 1 \times 9\,216$	1×1	1	$1 \times 1 \times 4\,096$	$(9\,216 + 1) \times 4\,096$
第 7 层	FC7	$1 \times 1 \times 4\,096$	1×1	1	$1 \times 1 \times 4\,096$	$(4\,096 + 1) \times 4\,096$
输出	FC8	$1 \times 1 \times 4\,096$	1×1	1	$1 \times 1 \times 1\,000$	$(4\,096 + 1) \times 1\,000$
总共						61 472 488

与 AlexNet 相比，ZFNet 的不同之处在于：① ZFNet 网络使用较小的掩码以保留更多原始像素信息，例如，AlexNet 第一个卷积层使用 11×11 大小的卷积核，而 ZFNet 使用 7×7 大小的卷积核；② ZFNet 网络只使用一个 GPU。ZFNet 各层的详细说明如下。

第 1 层包含卷积层 CONV1 和池化层 POOL1。卷积层 CONV1 利用 96 个大小为 $7 \times 7 \times 3$ 、步长为 2 的卷积核，对大小为 $224 \times 224 \times 3$ 的输入图像进行滤波，得到大小为 $110 \times 110 \times 96$ 的卷积结果。池化层 POOL1 利用大小为 3×3 、步长为 2 的池化窗口，对大小为 $110 \times 110 \times 96$ 的卷积输出结果进行池化，得到大小为 $55 \times 55 \times 96$ 的池化结果。

第 2 层包含卷积层 CONV2 和池化层 POOL2。卷积层 CONV2 利用 256 个大小为 $5 \times 5 \times 96$ 、步长为 2 的卷积核，对大小为 $55 \times 55 \times 96$ 的池化结果进行滤波，得到大小为 $26 \times 26 \times 256$ 的卷积结果。池化层 POOL2 利用大小为 3×3 、步长为 2 的池化窗口，对大小为 $26 \times 26 \times 256$ 的卷积输出结果进行池化，得到大小为 $13 \times 13 \times 256$ 的池化结果。



第3层是卷积层 CONV3，利用 384 个大小为 $3 \times 3 \times 256$ 、步长为 1 的卷积核，对大小为 $13 \times 13 \times 256$ 的池化结果进行滤波，得到大小为 $13 \times 13 \times 384$ 的卷积结果。

第4层是卷积层 CONV4，利用 384 个大小为 $3 \times 3 \times 384$ 、步长为 1 的卷积核，对大小为 $13 \times 13 \times 384$ 的卷积结果进行滤波，得到大小为 $13 \times 13 \times 384$ 的卷积结果。

第5层包含卷积层 CONV5 和池化层 POOL3。卷积层 CONV5 利用 256 个大小为 $3 \times 3 \times 384$ 、步长为 1 的卷积核，对大小为 $13 \times 13 \times 384$ 的卷积结果进行滤波，得到大小为 $13 \times 13 \times 256$ 的卷积结果。池化层 POOL 利用 256 个大小为 3×3 、步长为 2 的池化窗口，对大小为 $13 \times 13 \times 256$ 的卷积结果进行池化，得到大小为 $6 \times 6 \times 256$ 的卷积结果。

第6层是全连接层 FC6，拥有 4096 个节点。

第7层是全连接层 FC7，拥有 4096 个节点。

最后，输出层 (FC8) 有 1000 个节点。

ZFNet 与 AlexNet 非常相似，只是在某些方面做了细微的改变。通过参考 AlexNet 在 4.2 节的 Caffe 代码实现和 4.4 节的 TensorFlow 代码实现，读者只需进行相应的修改便可得到 ZFNet 的 Caffe 代码和 TensorFlow 代码，并按照有关命令进行实验。



第5章

卷积神经网络的应变模型

通常卷积神经网络要求输入图像具有固定的大小，难以灵活适应各种大小变化的实际情况。一种有效策略是采用空间金字塔池化网络 SPPNet，其主要特点是在最后一个卷积层和第一个全连接层之间插入了一个空间金字塔池化层。利用空间金字塔池化层，SPPNet 无须对输入图像进行裁剪或变形，就可以处理输入图像大小不同的情况。本章将介绍 SPPNet 的模型结构、Caffe 代码实现及说明、大规模图像分类案例及演示效果。

5.1 SPPNet 的模型结构

一般说来，深层卷积神经网络要求输入图像的大小是固定的（比如 224×224 ），这限制了输入图像的高宽比和大小。在实际应用时，卷积网络大多需要把输入图像裁剪或变形到固定大小，如图 5.1b 所示。然而，图 5.1a 表明，裁剪可能会造成目标缺失，变形又可能产生不必要的几何扭曲，进而导致识别准确率的下降。此外，由于目标大小可能任意变化，很难预定义一个相匹配的合适固定大小。

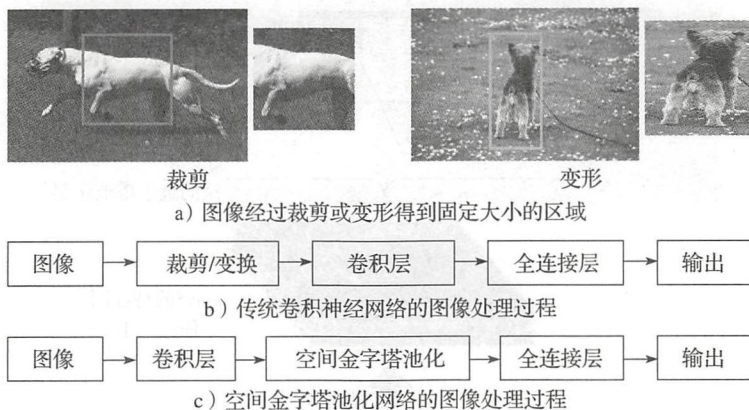


图 5.1

卷积神经网络要求输入固定大小的原因在于它包含两个部分：卷积层和后面的全连接



层。卷积层用滑动窗方式进行运算，输出表示激活值空间排列的特征图。事实上，卷积层不需要固定图像大小就能生成任意大小的特征图。另一方面，全连接层根据定义需要固定大小/长度的输入。因此，固定大小的限制仅仅来自于全连接层，存在于网络的一个更深阶段。引入空间金字塔池化（Spatial Pyramid Pooling, SPP）层^[110-111]，可以移除对输入图像固定大小的限制。如图 5.1c 所示，SPP 层加在最后一个卷积层之上，用来池化特征并产生固定长度的输出，然后该输出再被输入到全连接层（或其他分类器）。换句话说，SPP 层在一个网络层次较深的阶段（位于卷积层和全连接层之间）进行信息聚合，目的是避免开始时对裁剪或变形的需要。这种带有 SPP 层的卷积网络，称为 SPPNet、SPPnet 或 SPP-net^[112]。

空间金字塔池化，又称空间金字塔匹配（Spatial Pyramid Matching, SPM），作为一种词袋（Bag-of-Words, BoW）模型的扩展^[111]，是计算机视觉领域非常成功的方法之一。这种池化根据从细到粗的不同级别把图像划分成若干小块（即池化格），再聚合池化格中的局部特征。在卷积神经网络普及推广之前，SPP 在图像分类^[113-114]和检测^[115]的性能领先及竞赛获胜系统中一直都是非常关键的技术组成部分。SPP 具有以下几个引人注目的特征：① SPP 对任意输入大小都能够产生一个固定长度的输出，而滑动窗口池化却不能；② SPP 使用多级大小空间窗口，而滑动窗口池化只使用一个窗口大小；③ SPP 可以在不同尺度上提取特征并进行池化。这些特征使得 SPP 比滑动窗口池化具有更大的灵活性和更强的鲁棒性。

图 5.2 给出了一个 SPPNet 的网络结构，其基础是 ZFNet（详情可参考第 4 章），最后一个卷积层的卷积面个数记为 k 。这个 SPPNet 是把 ZFNet 的第一个全连接层替换为 SPP 层得到的。其中，SPP 层包含多级最大池化，如果第 l 级有一个池化格，那么第 l 级就输出一维的固定长度向量。SPP 层就是这些固定长度向量拼接成的全连接层，其最大优点在于能够用固定长度的向量表示不同尺度的空间信息。

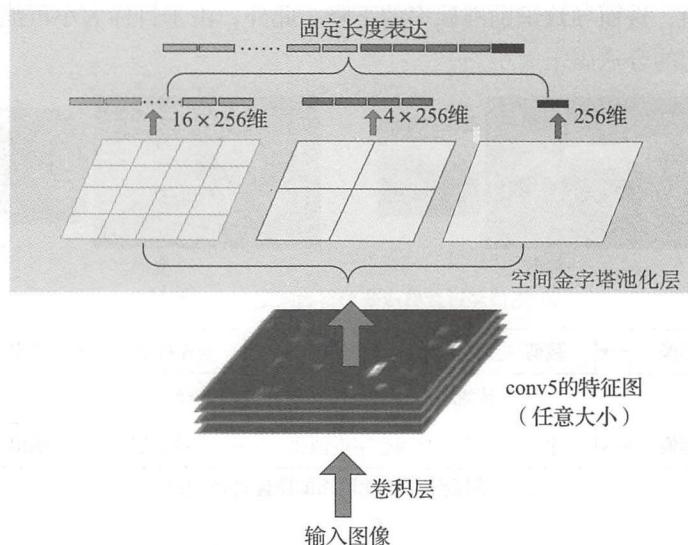


图 5.2 一个 SPPNet 的网络结构。其中，conv5 是最后一个卷积层，具有 256 个卷积核



理论上, SPPNet 可以采用反向传播方法进行训练, 而不用考虑输入图像的大小。但实际上在用 cuda-convnet^[46] 和 Caffe^[116] 对 SPPNet 进行 GPU 实现时, 最好还是运行在固定大小的输入图像上。为了利用 GPU 实现的优点, 同时保持空间金字塔池化的特性, 还需要在训练方法上做一些特殊处理, 具体的策略包括单尺度训练和多尺度训练两种方法。

单尺度训练是多尺度训练的基础, 主要是从图像中裁剪固定大小的输入(比如 224×224)来训练一个网络。裁剪的目的是扩增数据。对一幅给定大小的图像, 可以先计算空间金字塔池化格的大小。如果 conv5 的特征图大小为 $a \times a$ (比如 13×13), 那么实现一个 $n \times n$ 池化格的金字塔级, 所用滑动窗口的大小应为 $\text{win} = \lceil a/n \rceil$, 步长应为 $\text{str} = \lfloor a/n \rfloor$, 其中 $\lceil \cdot \rceil$ 和 $\lfloor \cdot \rfloor$ 是指天棚函数 (ceiling function) 和地板函数 (floor function)。对一个有 l 级的金字塔, 就需要实现 l 个这种层, 然后再把它们的 l 个输出拼接起来输入到全连接层 (fc6)。图 5.3 给出了一个用 cuda-convnet 形式实现 3 级金字塔构造的例子, 其中包含 3×3 、 2×2 和 1×1 池化格。单尺度训练的目的在于保证多级池化的特性。

多尺度训练的主要目的是处理任意大小的图像, 需要考虑至少两种不同的预定义大小, 比如 224×224 和 180×180 。注意, 180×180 区域并不是通过重新裁剪得到的, 而是把相应的 224×224 区域按比例缩小得到的。因此, 它们只是分辨率不同, 但内容相同。 180×180 区域被输入到另一个输入大小固定的 180×180 网络进行训练,

其中每层与原来的 224×224 网络具有完全相同的参数。不过, conv5 的特征图大小变为 $a \times a = 10 \times 10$, 且仍采用 $\text{win} = \lceil a/n \rceil$ 和 $\text{str} = \lfloor a/n \rfloor$ 去实现每一个金字塔池化级, 从而保证两个网络的空间金字塔池化层具有相同的固定长度。在训练过程中, 这两个共享参数的网络保持大小不变。它们的作用是联合实现输入大小可变的 SPPNet, 减少从一个网络到另一个网络的转换开销。实际的训练过程是先在一个网络上训练一次完整的迭代, 再转换到另一个网络 (保持所有的权值) 训练另一次完整的迭代, 不断循环直到收敛。

SPPNet 能够提高卷积网络在图像分类方面的性能。在 ImageNet 2012 数据集上的对比实验表明, 有四种卷积神经网络加入 SPP 层后与不加相比效果都有所提高^[46, 117-118]。在 Caltech101^[119] 和 VOC 2007^[120] 上 SPPNet 也都取得了领先的分类结果。除了分类, SPPNet 还可以用于目标检测。SPPNet 的一个早期版本发表在 ECCV 2014 上, 在参与 ILSVRC 2014 竞赛的 38 个队中, 取得目标检测第二名、图像分类第三名。关于目标检测方面的应用, 请读者进一步查阅相关文献 [112]。

[pool3 × 3] type=pool pool=max inputs=conv5 sizeX=5 stride=4	[pool2 × 2] type=pool pool=max inputs=conv5 sizeX=7 stride=6	[pool1 × 1] type=pool pool=max inputs=conv5 sizeX=13 stride=13
[fc6] type=fc outputs=4096 inputs=pool3 × 3, pool2 × 2, pool1 × 1		

图 5.3 一个用 cuda-convnet 形式实现 3 级金字塔构造的例子。其中, sizeX 是池化窗口的大小, conv5 的特征图大小是 13×13 , pool3 × 3、pool2 × 2 和 pool1 × 1 的池化格分别是 3×3 、 2×2 和 1×1



5.2 SPPNet 的 Caffe 代码实现及说明

关于 SPPNet 的 Caffe 大规模图像分类代码，可以从网址 <https://github.com/dsisds/caffe-SPPNet> 下载。如果读者需要另行进行目标检测，可以从网址 https://github.com/ShaoqingRen/SPP_net 下载。

为了方便读者使用，本书作者已对 SPPNet 的 Caffe 大规模图像分类代码进行了修改，并上传到网址 <https://github.com/TingZhang08/SPPNet>。其中包含 4 个文件：train_val_224.prototxt、train_val_180.prototxt、solver224.prototxt 和 solver180.prototxt。train_val_224.prototxt 和 train_val_180.prototxt 分别用来定义 SPPNet 网络的两种结构，它们的代码非常相似。solver224.prototxt 和 solver180.prototxt 用来配置网络训练的超参数。考虑到 SPPNet 与 AlexNet 和 ZFNet 的代码相似性，下面仅对 train_val_224.prototxt 文件中的卷积层 conv1 和空间金字塔模块 pool3 进行详细说明。

1. 卷积层 conv1

```
layer{
  name: "conv1"                # 名字
  type: "Convolution"
  bottom: "data"               # 前一层
  top: "conv1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  convolution_param {
    num_output: 96              # 卷积核的个数
    kernel_size: 7              # 卷积核的大小
    pad: 3
    stride: 2                   # 卷积核移动的步长
    weight_filler {
      type: "gaussian"          # 权值初始化方式
      std: 0.01
    }
    bias_term: false
  }
}
```

2. 空间金字塔模块 pool3

```
layer{
  name: "pool3_1"              # 名字，表示第一层级的池化
  type: "Pooling"
  bottom: "conv5"
  top: "pool3_1"
  pooling_param {
    pool: MAX
  }
}
```

```

        kernel_size: 13
        stride: 13
    }
}

layer{
    name: "pool3_2"           # 名字, 表示第二层级的池化
    type: "Pooling"
    bottom: "conv5"
    top: "pool3_2"
    pooling_param {
        pool: MAX
        kernel_size: 7
        stride: 6
    }
}

layer{
    name: "pool3_3"           # 名字, 表示第三层级的池化
    type: "Pooling"
    bottom: "conv5"
    top: "pool3_3"
    pooling_param {
        pool: MAX
        kernel_size: 5
        stride: 4
    }
}

layer{
    name: "pool3_4"           # 名字, 表示第四层级的池化
    type: "Pooling"
    bottom: "conv5"
    top: "pool3_4"
    pooling_param {
        pool: MAX
        kernel_size: 3
        stride: 2
    }
}

layer {
    name: "pool3"             # 名字, 表示对四个层级的池化结果进行拼接
    bottom: "pool3_1_flatt"
    bottom: "conv3_2_flatt"
    bottom: "pool3_3_flatt"
    bottom: "pool3_4_flatt"
    top: "pool3"
    type: "Concat"
    concat_param {
        axis: 1
    }
}

```

5.3 SPPNet 的大规模图像分类案例及演示效果

本节描述一个利用 SPPNet 在 Caffe 框架下进行大规模图像分类的案例，其中用到的 ImageNet 数据集可以根据表 1.2 提供的地址下载。注意，这个数据集的图像是 .jpg 格式的，需要参考 3.6.1 节转换成 Caffe 支持的 LEVELDB 格式。此外，还需根据方框 5.1 编写两个求解器配置文件 solver224.prototxt 和 solver180.prototxt，用于设置训练算法、学习率和迭代次数等内容。其中，对于 solver180.prototxt，只需把 solver224.prototxt 文件中的“train_val_224.prototxt”替换为“train_val_180.prototxt”。

方框 5.1 solver224.prototxt 文件中的超参数设置情况

```
net: "F:/YongD/caffe-windows-master/models/caffe-SPPNet-master/examples/
imagenet/train_val_224.prototxt"
test_iter: 1000
test_interval: 10000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 200000
display: 10000
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 450000
snapshot_prefix: "F:/YongD/caffe-windows-master/models/caffe-SPPNet-master/
examples/imagenet/caffe_train"
solver_mode: GPU
```

在编写好 solver224.prototxt 和 solver180.prototxt 之后，就可依次使用 224×224 和 180×180 大小的图像对 SPPNet 进行训练和验证。具体步骤为，首先进入 DOS 命令窗口，输入如图 5.4 所示的命令运行程序，对一个网络进行训练和验证。程序运行过程中，随着迭代次数的增加，网络的训练损失和准确率也发生变化。如图 5.5 所示，中间结果在迭代次数为 160 000 时，学习率为 0.01，训练损失为 5.463 57，top-1 训练准确率为 6.25%，top-5 训练准确率约为 15.63%，top-1 验证准确率约为 8.29%，top-5 验证准确率约为 21.86%。如图 5.6 所示，在程序运行结束时，训练损失为 1.971 77，top-1 验证准确率约为 42.85%，top-5 验证准确率约为 67.84%。

然后，利用训练好的网络初始化另一个网络，并进行训练和验证。具体步骤为，在命令窗口输入如图 5.7 所示的命令运行程序。程序运行过程中，随着迭代次数的增加，网络的训练损失和准确率也发生变化。如图 5.8 所示，中间结果在迭代次数为 330 000 时，学习率为 0.001，训练损失为 2.632 89，top-1 训练准确率约为 42.19%，top-5 训练准确率约为 64.06%，top-1 验证准确率约为 35.38%，top-5 验证准确率约为 60.72%。如图 5.9 所示，在程序运行结束时，训练损失为 2.143 27，top-1 验证准确率约为 41.99%，top-5 验证准确率约为 67.32%。

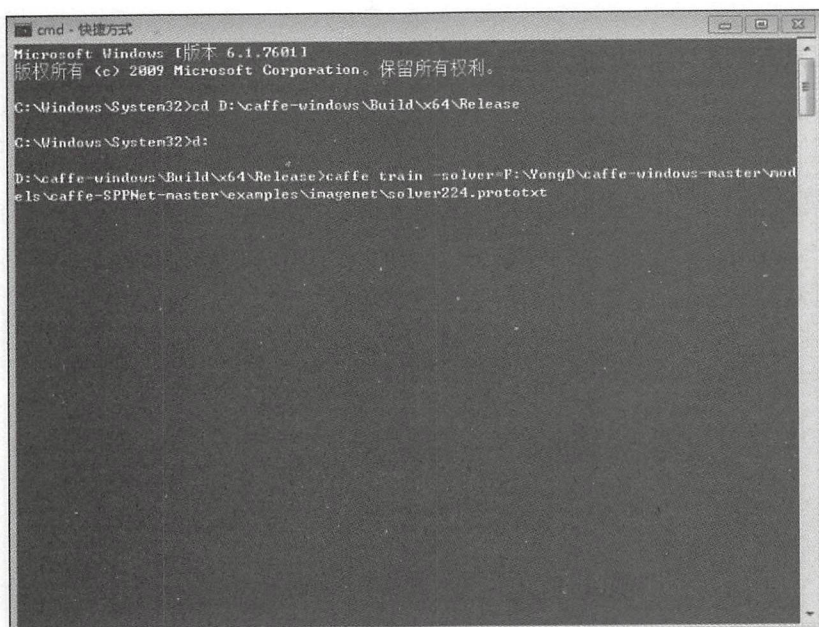


图 5.4 SPPNet 大规模图像分类案例的网络训练命令

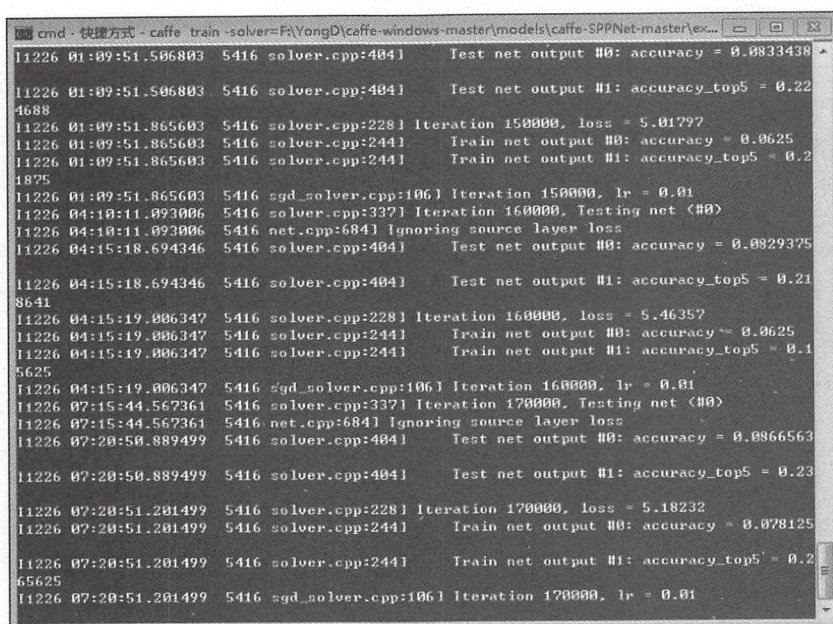


图 5.5 SPPNet 大规模图像分类案例训练网络的中间结果

注意，这里只对网络各训练和验证了一次。若读者有兴趣还可以对它们进行多次训练和验证，但应使用新训练好的网络初始化下一次要训练的网络。

```

I1228 21:31:01.320680 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:31:33.120499 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:32:16.872923 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:32:42.458672 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:33:12.038616 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:33:43.008709 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:34:08.545953 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:34:37.552582 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:35:02.144989 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:35:30.826829 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:35:58.641679 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:36:24.069723 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:36:51.322971 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:37:16.735415 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:40:18.329761 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:40:45.193008 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:41:13.366657 7212 blocking_queue.cpp:501 Waiting for data
I1228 21:41:13.663058 5416 solver.cpp:4541 Snapshotting to binary proto file F:\YongD\c
affe-windows-master\models\caffe-SPPNet-master\examples\imagenet\caffe_train_iter_450000
.caffemodel
I1228 21:41:16.798663 5416 sgdsolver.cpp:2731 Snapshotting solver state to binary prot
o file F:\YongD\caffe-windows-master\models\caffe-SPPNet-master\examples\imagenet\caffe_
train_iter_450000.solverstate
I1228 21:41:17.750265 5416 solver.cpp:3171 Iteration 450000, loss = 1.97177
I1228 21:41:17.750265 5416 solver.cpp:3371 Iteration 450000, Testing net (#0)
I1228 21:41:17.750265 5416 net.cpp:6841 Ignoring source layer loss
I1228 21:43:40.194115 5416 solver.cpp:4041 Test net output #0: accuracy = 0.428469
I1228 21:43:40.194115 5416 solver.cpp:4041 Test net output #1: accuracy_top5 = 0.67
8422
I1228 21:43:40.194115 5416 solver.cpp:3221 Optimization Done.
I1228 21:43:40.194115 5416 caffe.cpp:2231 Optimization Done.

D:\caffe-windows\Build\x64\Release>

```

图 5.6 SPPNet 大规模图像分类案例训练网络的最终结果

```

Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Windows\System32>cd D:\caffe-windows\Build\x64\Release

C:\Windows\System32>

D:\caffe-windows\Build\x64\Release>caffe train -solver=F:\YongD\caffe-windows-master\models\caffe-SPPNet-master\examples\imagenet\solver180.prototxt -weights=F:\YongD\caffe-windows-master\models\caffe-SPPNet-master\examples\imagenet\caffe_train_iter_450000.caffemodel

```

图 5.7 SPPNet 大规模图像分类案例的网络训练命令


```
cmd - 快捷方式
03125
10103 12:49:11.597712 12712 sgd_solver.cpp:1061 Iteration 320000, lr = 0.001
10103 14:13:13.332083 12712 solver.cpp:3371 Iteration 330000, Testing net (#0)
10103 14:13:13.332083 12712 net.cpp:6841 Ignoring source layer loss
10103 14:15:36.529273 12712 solver.cpp:4041 Test net output #0: accuracy = 0.353797
10103 14:15:36.530273 12712 solver.cpp:4041 Test net output #1: accuracy_top5 = 0.60
7156
10103 14:15:36.674281 12712 solver.cpp:2281 Iteration 330000, loss = 2.63289
10103 14:15:36.674281 12712 solver.cpp:2441 Train net output #0: accuracy = 0.421875
10103 14:15:36.674281 12712 solver.cpp:2441 Train net output #1: accuracy_top5 = 0.6
40625
10103 14:15:36.675282 12712 sgd_solver.cpp:1061 Iteration 330000, lr = 0.001
10103 15:39:38.425653 12712 solver.cpp:3371 Iteration 340000, Testing net (#0)
10103 15:39:38.425653 12712 net.cpp:6841 Ignoring source layer loss
10103 15:42:01.495836 12712 solver.cpp:4041 Test net output #0: accuracy = 0.351125
10103 15:42:01.495836 12712 solver.cpp:4041 Test net output #1: accuracy_top5 = 0.60
7047
10103 15:42:01.643846 12712 solver.cpp:2281 Iteration 340000, loss = 3.47057
10103 15:42:01.643846 12712 solver.cpp:2441 Train net output #0: accuracy = 0.390625
10103 15:42:01.643846 12712 solver.cpp:2441 Train net output #1: accuracy_top5 = 0.5
3125
10103 15:42:01.643846 12712 sgd_solver.cpp:1061 Iteration 340000, lr = 0.001
10103 17:06:03.385216 12712 solver.cpp:3371 Iteration 350000, Testing net (#0)
10103 17:06:03.385216 12712 net.cpp:6841 Ignoring source layer loss
10103 17:08:26.552404 12712 solver.cpp:4041 Test net output #0: accuracy = 0.366437
10103 17:08:26.552404 12712 solver.cpp:4041 Test net output #1: accuracy_top5 = 0.62
2547
10103 17:08:26.697413 12712 solver.cpp:2281 Iteration 350000, loss = 3.23589
10103 17:08:26.697413 12712 solver.cpp:2441 Train net output #0: accuracy = 0.296875
10103 17:08:26.697413 12712 solver.cpp:2441 Train net output #1: accuracy_top5 = 0.5
```

图 5.8 SPPNet 大规模图像分类案例训练网络的中间结果

```
cmd - 快捷方式
1031
10104 04:38:52.480832 12712 solver.cpp:2281 Iteration 430000, loss = 1.84925
10104 04:38:52.480832 12712 solver.cpp:2441 Train net output #0: accuracy = 0.53125
10104 04:38:52.480832 12712 solver.cpp:2441 Train net output #1: accuracy_top5 = 0.7
65625
10104 04:38:52.480832 12712 sgd_solver.cpp:1061 Iteration 430000, lr = 0.0001
10104 06:02:54.018191 12712 solver.cpp:3371 Iteration 440000, Testing net (#0)
10104 06:02:54.019191 12712 net.cpp:6841 Ignoring source layer loss
10104 06:05:17.176380 12712 solver.cpp:4041 Test net output #0: accuracy = 0.426344
10104 06:05:17.176380 12712 solver.cpp:4041 Test net output #1: accuracy_top5 = 0.67
7984
10104 06:05:17.326388 12712 solver.cpp:2281 Iteration 440000, loss = 1.90187
10104 06:05:17.326388 12712 solver.cpp:2441 Train net output #0: accuracy = 0.609375
10104 06:05:17.326388 12712 solver.cpp:2441 Train net output #1: accuracy_top5 = 0.7
8125
10104 06:05:17.326388 12712 sgd_solver.cpp:1061 Iteration 440000, lr = 0.0001
10104 07:29:18.787744 12712 solver.cpp:4541 Snapshotting to binary proto file F:\YongD\c
affe-windows-master\models\caffe-SPPNet-master/examples\imagenet\caffe_train_iter_450000
.caffemodel
10104 07:29:21.975926 12712 sgd_solver.cpp:2731 Snapshotting solver state to binary prot
o file F:\YongD\caffe-windows-master\models\caffe-SPPNet-master/examples\imagenet\caffe_
train_iter_450000.solverstate
10104 07:29:22.939981 12712 solver.cpp:3171 Iteration 450000, loss = 2.14327
10104 07:29:22.939981 12712 solver.cpp:3371 Iteration 450000, Testing net (#0)
10104 07:29:22.939981 12712 net.cpp:6841 Ignoring source layer loss
10104 07:31:45.633142 12712 solver.cpp:4041 Test net output #0: accuracy = 0.419875
10104 07:31:45.633142 12712 solver.cpp:4041 Test net output #1: accuracy_top5 = 0.67
3219
10104 07:31:45.634142 12712 solver.cpp:3221 Optimization Done.
10104 07:31:45.634142 12712 caffe.cpp:2231 Optimization Done.
D:\caffe-windows\Build\x64\Release>
```

图 5.9 SPPNet 大规模图像分类案例训练网络的最终结果

卷积神经网络的加深模型

随着深度学习研究的推进，卷积神经网络开始不断向纵深化方向发展，相继出现了结构很深的 VGGNet 和 GoogLeNet 等新模型。VGGNet 验证了增加网络的深度确实可以提高网络的性能，它在 2014 年 ILSVRC 挑战赛中赢得了定位任务冠军和分类任务亚军。GoogLeNet 则在加深网络结构的同时，还提出了称为 “Inception” 的新模块，并赢得了 2014 年 ILSVRC 挑战赛的分类任务冠军。本章主要介绍卷积神经网络的两种加深模型——VGGNet 和 GoogLeNet，包括它们的模型结构、TensorFlow 代码实现及说明，以及分别用它们实现的 CIFAR-10 物体图像分类案例和 Oxford-17 鲜花图像分类案例。

6.1 结构加深的卷积网络 VGGNet

作为卷积神经网络的突破模型，AlexNet 的出色工作表明，通过增加网络的深度可以提升网络的性能。VGGNet 是沿着这一思路对卷积网络的结构逐步加深建立起来的。VGGNet 曾经是一种非常深的卷积网络模型^[110]，其命名依据是 2014 年参加 ILSVRC 竞赛的 “VGG” 队。VGGNet 的核心思想是利用较小的卷积核来增加网络的深度，有两种基本类型：一种称为 VGGNet-16，另一种称为 VGGNet-19。

6.1.1 VGGNet 的模型结构

为了利用加深结构的思路来提升性能，VGGNet 全部使用 3×3 的卷积核和 2×2 的池化核。VGGNet 是在比较 6 种不同加深结构的基础上提出来的卷积网络模型。这 6 种加深结构分别标记为 A、A-LRN、B、C、D 和 E，详细描述如表 6.1 所示，参数个数如表 6.2 所示。从表 6.1 可以看出：

- 1) 模型 A：拥有 8 个卷积层（核大小均为 3×3 ）、5 个最大池化层和 3 个全连接层，共有 11 个权值层。
- 2) 模型 A-LRN：与模型 A 几乎相同，区别在于第 1 个卷积层之后加了一个局部响应

归一化层 (LRN)。实验表明, LRN 会增加内存消耗和计算时间, 且无助于改善性能, 所以在 B、C、D 和 E 中被取消。

3) 模型 B: 拥有 10 个卷积层 (核大小均为 3×3)、5 个最大池化层和 3 个全连接层, 共有 13 个权值层。

4) 模型 C: 拥有 13 个卷积层 (10 个核大小为 3×3 , 3 个核大小为 1×1)、5 个最大池化层和 3 个全连接层, 共有 16 个权值层。

5) 模型 D: 拥有 13 个卷积层 (核大小均为 3×3)、5 个最大池化层和 3 个全连接层, 共有 16 个权值层。

6) 模型 E: 拥有 16 个卷积层 (核大小均为 3×3)、5 个最大池化层和 3 个全连接层, 共有 19 个权值层。

表 6.1 与 VGGNet 有关的 6 种加深结构

A	A-LRN	B	C	D (VGGNet-16)	E (VGGNet-19)
11 个权值层	11 个权值层	13 个权值层	16 个权值层	16 个权值层	19 个权值层
输入 (224×224 RGB 图像)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
最大池化层					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
最大池化层					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
最大池化层					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
最大池化层					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
最大池化层					
FC-4096					
FC-4096					
FC-1000					
软最大输出层					

表 6.2 加深结构的参数个数

(单位：百万)

网络	A、A-LRN	B	C	D	E
参数个数	133	133	134	138	144

上述 6 种加深结构都使用 ReLU 激活函数，其中结构 D 就是“VGGNet-16”，结构 E 就是“VGGNet-19”。VGGNet-19 与 VGGNet-16 的区别在于多了 3 个卷积层。

6.1.2 VGGNet 的 TensorFlow 代码实现及说明

本节利用 TensorFlow 框架实现 VGGNet-16 对物体图像数据集 CIFAR-10 的分类，主要包括 4 个程序：cifar10_input.py、cifar10.py、cifar10_train.py 和 cifar10_eval.py。其中，cifar10_input.py 定义了网络的数据输入情况，涉及对 CIFAR-10 数据的下载、读取、分块和输入等操作，为训练和测试做准备。cifar10.py 定义了网络的结构情况，包括网络训练结构和测试结构。cifar10_train.py 定义了网络的训练情况，涉及训练数据的读取和存放路径、训练参数的设置、训练过程的打印显示和训练模型的保存等内容。cifar10_eval.py 定义了网络的测试情况，涉及测试数据的读取和存放路径、已训练模型的调用和测试过程的打印显示等内容。下面分别对这 4 个程序的代码进行详细说明。

1. cifar10_input.py 的代码及说明

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import os
from six.moves import xrange          # pylint: disable=redefined-builtin
import tensorflow as tf
IMAGE_SIZE = 32
NUM_CLASSES = 10
NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN = 50000
NUM_EXAMPLES_PER_EPOCH_FOR_EVAL = 10000
def read_cifar10(filename_queue):      # 从 CIFAR-10 数据文件中读取和解析样本
    class CIFAR10Record(object):
        pass
    result = CIFAR10Record()
    label_bytes = 1
    result.height = 32
    result.width = 32
    result.depth = 3
    image_bytes = result.height * result.width * result.depth
    record_bytes = label_bytes + image_bytes    # 每个记录由一个标签和一个图像组成
    reader = tf.FixedLengthRecordReader(record_bytes=record_bytes)    # 读取记录
    result.key, value = reader.read(filename_queue)    # 读取记录，从 filename_
                                                    # queue 获取文件名
    record_bytes = tf.decode_raw(value, tf.uint8)    # 字符串转换成 uint8 的向量
    result.label = tf.cast(tf.slice(record_bytes, [0], [label_bytes]),
```



```

tf.int32) # 把标签从 uint8 转成 int32
depth_major = tf.reshape(tf.slice(record_bytes, [label_bytes], [image_bytes]),
                           [result.depth, result.height, result.width])

# reshape 图像大小
result.uint8image = tf.transpose(depth_major, [1, 2, 0])
# 将图像 [depth, height, width] 转为 [height, width, depth]
return result

def _generate_image_and_label_batch(image, label, min_queue_examples, batch_
size, shuffle):
    # 构建图像和标签的 batch 队列
    num_preprocess_threads = 16
    if shuffle: # 创建和打乱队列, 然后读取 batch_size 图像 + 标签队列
        images, label_batch = tf.train.shuffle_batch(
            [image, label], batch_size=batch_size, num_threads=num_preprocess_
            threads,
            capacity=min_queue_examples + 3 * batch_size,
            min_after_dequeue=min_queue_examples)
    else:
        images, label_batch = tf.train.batch(
            [image, label], batch_size=batch_size, num_threads=num_preprocess_
            threads, capacity=min_queue_examples + 3 * batch_size)
    tf.summary.image('images', images) # 显示训练图像的可视化工具
    return images, tf.reshape(label_batch, [batch_size])

def distorted_inputs(data_dir, batch_size): # 该函数生成扭曲后的图像, 用于扩充训练集
    filenames = [os.path.join(data_dir, 'data_batch%d.bin' % i) for i in
xrange(1, 6)]
    for f in filenames:
        if not tf.gfile.Exists(f): # 如果目录索引不存在, 则报错
            raise ValueError('Failed to find file: ' + f)
        filename_queue = tf.train.string_input_producer(filenames)
        # 创建文件名队列, 用于读取数据
        read_input = read_cifar10(filename_queue) # 从文件名队列中的文件读取样本
        reshaped_image = tf.cast(read_input.uint8image, tf.float32)
        # 将数据转换为 float32 类型
        height = IMAGE_SIZE
        width = IMAGE_SIZE
        distorted_image = tf.image.random_flip_left_right(reshaped_image)
        # 图像随机左右翻转
        distorted_image = tf.image.random_brightness(distorted_image, max_delta=63)
        # 亮度变化
        distorted_image = tf.image.random_contrast(distorted_image, lower=0.2, upper=1.8)
        # 对比度变化
        float_image = tf.image.per_image_standardization(distorted_image)
        # 将整幅图像标准化 (减均值除方差)
        min_fraction_of_examples_in_queue = 0.4
        min_queue_examples = int(NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN*min_fraction_of_
examples_in_queue)
        print ('Filling queue with %d CIFAR images before starting to train. ' 'This
will take a few minutes.' % min_queue_examples)
        return _generate_image_and_label_batch(float_image, read_input.label,
min_queue_examples, batch_size, shuffle=True)

def inputs(eval_data, data_dir, batch_size): # 读取测试数据并生成 batch 队列

```

```

if not eval_data:
    filenames = [os.path.join(data_dir, 'data_batch%d.bin' % i) for i in
                  xrange(1, 6)]
    num_examples_per_epoch = NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN
else:
    filenames = [os.path.join(data_dir, 'test_batch.bin')]
    num_examples_per_epoch = NUM_EXAMPLES_PER_EPOCH_FOR_EVAL
for f in filenames:
    if not tf.gfile.Exists(f):
        raise ValueError('Failed to find file: ' + f)
    filename_queue = tf.train.string_input_producer(filenames)
    # 创建文件名队列，用于读取数据
    read_input = read_cifar10(filename_queue) # 从文件名队列中的文件读取样本
    reshaped_image = tf.cast(read_input.uint8image, tf.float32)
    # 将数据转换为 float32 类型
    height = IMAGE_SIZE
    width = IMAGE_SIZE
    float_image = tf.image.per_image_standardization(reshaped_image)
    # 将整幅图像标准化（减均值除方差）
    min_fraction_of_examples_in_queue = 0.4
    min_queue_examples = int(num_examples_per_epoch * min_fraction_of_examples_in_queue)
    return _generate_image_and_label_batch(float_image, read_input.label,
                                            min_queue_examples, batch_size, shuffle=False)

```

2. cifar10.py 的代码及说明

```

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import gzip
import os
import re
import sys
import tarfile
from six.moves import urllib
import tensorflow as tf
import cifar10_input
FLAGS = tf.app.flags.FLAGS
tf.app.flags.DEFINE_integer('batch_size', 128, """Number of images to process
in a batch.""")
tf.app.flags.DEFINE_string('data_dir', './cifar10/cifar10_data',
                           """Path to the CIFAR-10 data directory.""")
tf.app.flags.DEFINE_boolean('use_fp16', False, """Train the model using
fp16.""")
IMAGE_SIZE = cifar10_input.IMAGE_SIZE
NUM_CLASSES = cifar10_input.NUM_CLASSES
NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN =
    cifar10_input.NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN
NUM_EXAMPLES_PER_EPOCH_FOR_EVAL =
    cifar10_input.NUM_EXAMPLES_PER_EPOCH_FOR_EVAL

```

```

MOVING_AVERAGE_DECAY = 0.9999          # 移动平均线的衰变
NUM_EPOCHS_PER_DECAY_1 = 75.0          # 两次学习率衰减之间的迭代次数
LEARNING_RATE_DECAY_FACTOR = 0.01      # 学习率衰减率
INITIAL_LEARNING_RATE = 0.01           # 初始学习率
WD = 0.001                             # 权值衰减(正则化)
TOWER_NAME = 'tower'
DATA_URL = 'http://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz'
# cifar10 数据下载地址
def _activation_summary(x):              # 创建激活总结, 提供可视化
    tensor_name = re.sub('%s_[0-9]*/' % TOWER_NAME, '', x.op.name)
    # 正则替换字符串
    tf.summary.histogram(tensor_name + '/activations', x)  # 参数在训练中可视化
    tf.summary.scalar((tensor_name + '/sparsity', tf.nn.zero_fraction(x))
    # 标量在训练中可视化
def _variable_on_cpu(name, shape, initializer):  # 创建存储在 CPU 内存中的变量
    with tf.device('/cpu:0'):
        dtype = tf.float16 if FLAGS.use_fp16 else tf.float32
        var = tf.get_variable(name, shape, initializer=initializer, dtype=dtype)
    return var
def distorted_inputs():                  # 该函数生成扭曲后的图像, 用于扩充训练集
    if not FLAGS.data_dir:               # 如果目录索引不存在, 则报错
        raise ValueError('Please supply a data_dir')
    data_dir = os.path.join(FLAGS.data_dir, 'cifar-10-batches-bin')
    images, labels = cifar10_input.distorted_inputs(data_dir=data_dir, batch_size=FLAGS.batch_size)
    if FLAGS.use_fp16: # 如果 FLAGS.use_fp16 为真时, 转换 images 和 labels 格式为 float16
        images = tf.cast(images, tf.float16)
        labels = tf.cast(labels, tf.float16)
    return images, labels
def inputs(eval_data):                  # 测试数据的输入函数
    if not FLAGS.data_dir:              # 如果目录索引不存在, 则报错
        raise ValueError('Please supply a data_dir')
    data_dir = os.path.join(FLAGS.data_dir, 'cifar-10-batches-bin')
    images, labels = cifar10_input.inputs(eval_data=eval_data, data_dir=data_dir, batch_size=FLAGS.batch_size)
    if FLAGS.use_fp16: # 如果 FLAGS.use_fp16 为真时, 换 images 和 labels 格式为 float16
        images = tf.cast(images, tf.float16)
        labels = tf.cast(labels, tf.float16)
    return images, labels
def conv_op(input_op, name, kw, kh, n_out, dw, dh): # 定义卷积层操作
    n_in = input_op.get_shape()[-1].value
    with tf.variable_scope(name) as scope:
        kernel = _variable_on_cpu('weights', shape=[kh, kw, n_in, n_out],
                                   initializer=tf.contrib.layers.xavier_initializer_conv2d())
        conv = tf.nn.conv2d(input_op, kernel, [1, dh, dw, 1], padding='SAME')
        biases = _variable_on_cpu('bias', [n_out], tf.constant_initializer(0.0))
        z = tf.reshape(tf.nn.bias_add(conv, biases), conv.get_shape())
        activation = tf.nn.relu(z, name=scope.name)
    return activation
def fc_op(input_op, name, n_out):
    n_in = input_op.get_shape()[-1].value
    with tf.variable_scope(name) as scope:

```



```

        kernel = _variable_on_cpu('weights', shape=[n_in, n_out], initializer=tf.\
contrib.layers.xavier_initializer())
        biases = _variable_on_cpu('biases', [n_out], tf.constant_\
initializer(0.1))
        activation = tf.nn.relu_layer(input_op, kernel, biases,
name=scope.name)
    return activation

def mpool_op(input_op, name, kh, kw, dh, dw):
    # 定义最大池化层
    return tf.nn.max_pool(input_op, ksize=[1, kh, kw, 1], strides=[1, dh, dw,
1], padding='SAME', name=name)

def inference(images, training=True):
    # 定义 VGGNet 的结构
    dropout_keep_prob = 0.7 if training else 1.0
    conv1_1 = conv_op(images, name="conv1_1", kh=3, kw=3, n_out=16, dh=1, dw=1)
    conv1_2 = conv_op(conv1_1, name="conv1_2", kh=3, kw=3, n_out=16, dh=1, dw=1)
    pool1 = mpool_op(conv1_2, name="pool1", kh=2, kw=2, dh=2, dw=2)
    conv2_1 = tf.nn.dropout(conv_op(pool1, name="conv2_1", kh=3, kw=3, n_out=32,
dh=1, dw=1), dropout_keep_prob)
    conv2_2 = tf.nn.dropout(conv_op(conv2_1, name="conv2_2", kh=3, kw=3, n_\
out=32, dh=1, dw=1), dropout_keep_prob)
    pool2 = mpool_op(conv2_2, name="pool2", kh=2, kw=2, dh=2, dw=2)
    conv3_1 = tf.nn.dropout(conv_op(pool2, name="conv3_1", kh=3, kw=3, n_\
out=64, dh=1, dw=1), dropout_keep_prob)
    conv3_2 = tf.nn.dropout(conv_op(conv3_1, name="conv3_2", kh=3, kw=3, n_\
out=64, dh=1, dw=1), dropout_keep_prob)
    pool3 = mpool_op(conv3_2, name="pool3", kh=2, kw=2, dh=2, dw=2)
    conv4_1 = tf.nn.dropout(conv_op(pool3, name="conv4_1", kh=3, kw=3, n_\
out=128, dh=1, dw=1), dropout_keep_prob)
    conv4_2 = tf.nn.dropout(conv_op(conv4_1, name="conv4_2", kh=3, kw=3, n_\
out=128, dh=1, dw=1), dropout_keep_prob)
    conv4_3 = tf.nn.dropout(conv_op(conv4_2, name="conv4_3", kh=3, kw=3, n_\
out=128, dh=1, dw=1), dropout_keep_prob)
    pool4 = mpool_op(conv4_3, name="pool4", kh=2, kw=2, dh=2, dw=2)
    shp = pool4.get_shape()
    flattened_shape = shp[1].value * shp[2].value * shp[3].value
    resh1 = tf.reshape(pool4, [-1, flattened_shape], name="resh1") # 平铺展开层
    fc6 = fc_op(resh1, name="fc6", n_out=1024) # 全连接层
    fc6_drop = tf.nn.dropout(fc6, dropout_keep_prob, name="fc6_drop")
    # 使用 dropout 函数
    fc7 = fc_op(fc6_drop, name="fc7", n_out=1024) # 全连接层
    fc7_drop = tf.nn.dropout(fc7, dropout_keep_prob, name="fc7_drop")
    # 使用 dropout 函数
    fc8 = fc_op(fc7_drop, name="fc8", n_out=10) # 全连接层
    softmax = tf.nn.softmax(fc8) # 使用 softmax 函数
    predictions = tf.argmax(softmax, 1)
    return softmax, predictions, fc8

def loss(logits, labels):
    # 定义损失函数
    labels = tf.cast(labels, tf.int64)
    cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels =
labels, logits = logits,
name='cross_entropy_per_example') # 计算批次总的交叉熵损失
    cross_entropy_mean = tf.reduce_mean(cross_entropy, name='cross_entropy')
    # 计算平均值

```

```

    tf.add_to_collection('losses', cross_entropy_mean)
    # 将 cross_entropy_mean 以 losses 存储在收集器中
    return tf.add_n(tf.get_collection('losses'), name='total_loss')
def _add_loss_summaries(total_loss):    # 添加损失总结
    loss_averages = tf.train.ExponentialMovingAverage(0.9, name='avg')
    # 计算损失移动平均值和总损失
    losses = tf.get_collection('losses')
    loss_averages_op = loss_averages.apply(losses + [total_loss])
    for l in losses + [total_loss]:
        tf.summary.scalar(l.op.name + ' (raw)', l)
        tf.summary.scalar(l.op.name, loss_averages.average(l))
    return loss_averages_op
def train(total_loss, global_step):    # 训练 VGGNet 模型
    num_batches_per_epoch = NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN / FLAGS.batch_size
    decay_steps = int(num_batches_per_epoch * NUM_EPOCHS_PER_DECAY_1)
    decayed_learning_rate = tf.train.exponential_decay(INITIAL_LEARNING_RATE,
        global_step, decay_steps, LEARNING_RATE_DECAY_FACTOR, staircase=True)
    # 衰减学习
    lr=decayed_learning_rate
    tf.summary.scalar('learning_rate', lr)
    loss_averages_op = _add_loss_summaries(total_loss)
    with tf.control_dependencies([loss_averages_op]):    # 计算梯度
        opt = tf.train.MomentumOptimizer(lr, 0.9)
        grads = opt.compute_gradients(total_loss)
    apply_gradient_op = opt.apply_gradients(grads, global_step=global_step)
    # 应用梯度
    for var in tf.trainable_variables():    # 添加可修改变量的直方图
        tf.summary.histogram(var.op.name, var)
    for grad, var in grads:    # 添加梯度的直方图
        if grad is not None:
            tf.summary.histogram(var.op.name + '/gradients', grad)
    variable_averages = tf.train.ExponentialMovingAverage(MOVING_AVERAGE_1,
        DECAAY, global_step)
    variables_averages_op = variable_averages.apply(tf.trainable_variables())
    with tf.control_dependencies([apply_gradient_op, variables_averages_op]):
        train_op = tf.no_op(name='train')
    return train_op
def evaluate_op(predictions, labels):    # 计算准确率
    correct = tf.nn.in_top_k(predictions, labels, 1)
    total_correct = tf.reduce_sum(tf.cast(correct, tf.int32))
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
    return accuracy, total_correct
def maybe_download_and_extract():    # 可能用到的下载和解压函数
    dest_directory = FLAGS.data_dir
    if not os.path.exists(dest_directory):
        os.makedirs(dest_directory)
    filename = DATA_URL.split('/')[-1]
    filepath = os.path.join(dest_directory, filename)
    if not os.path.exists(filepath):
        def _progress(count, block_size, total_size):
            sys.stdout.write('\r>> Downloading %s %.1f%%' % (filename,
                # 输出不换行

```

```

        float(count * block_size) / float(total_size) * 100.0))
    sys.stdout.flush()      # 刷新 stdout
    filepath, _ = urllib.request.urlretrieve(DATA_URL, filepath,
        _progress)      # 将数据下载到本地
    print()
    statinfo = os.stat(filepath) # 根据路径读取文件的相关属性，并用 stat 模块处理
    print('Successfully downloaded', filename, statinfo.st_size, 'bytes.')
    tarfile.open(filepath, 'r:gz').extractall(dest_directory)

```

3. cifar10_train.py 的代码及说明

```

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from datetime import datetime
import os.path
import time
import numpy as np
from six.moves import xrange
import tensorflow as tf
import cifar10
FLAGS = tf.app.flags.FLAGS
tf.app.flags.DEFINE_string('train_dir', './cifar10/16_VGG/train',
    """Directory where to write event logs """)
tf.app.flags.DEFINE_integer('max_steps', 50000, """Number of batches to
run.""")
tf.app.flags.DEFINE_boolean('log_device_placement', False,
    """Whether to log device placement.""")
tf.app.flags.DEFINE_string('checkpoint_dir', './cifar10/16_VGG/checkpoint',
    """Directory where to write checkpoints""")
tf.app.flags.DEFINE_boolean('from_checkpoint', False, """Whether to launch
from checkpoint""")
def train(): # 训练网络
    with tf.Graph().as_default():
        global_step = tf.Variable(0, trainable=False) # 定义变量 global_step 并置为 0
        images, labels = cifar10.distorted_inputs() # 获取 CIFAR-10 的图像和标签
        images = tf.image.resize_images(images, [128, 128]) # 调整图像大小
        softmax, predictions, logits = cifar10.inference(images, training=True)
        # 构建预测计算图
        loss = cifar10.loss(logits, labels) # 计算损失
        accuracy, total_correct = cifar10.evaluate_op(softmax, labels)
        # 计算训练准确率
        train_op = cifar10.train(loss, global_step)
        # 用一批样本构建模型训练图并更新模型参数
        saver = tf.train.Saver(tf.all_variables(), max_to_keep=5) # 保存训练权值
        summary_op = tf.summary.merge_all() # 根据汇总集合构建汇总操作
        init = tf.initialize_all_variables() # 创建初始化操作
        sess = tf.Session(config=tf.ConfigProto(log_device_placement=FLAGS.\
            log_device_placement)) # 开始运行
        if FLAGS.from_checkpoint:

```



```

tf.train.Saver().restore(sess,
    '/home/roshan/class_models/CIFAR10/14_VGG/checkpoint/model.\
    ckpt-12000')
else:
    sess.run(init)
tf.train.start_queue_runners(sess=sess) # 启动输入队列线程, 填充样本到队列中
summary_writer = tf.summary.FileWriter(FLAGS.train_dir, sess.graph)
# 将运行结果写入磁盘
for step in xrange(FLAGS.max_steps):
    start_time = time.time()
    _, loss_value, training_acc = sess.run([train_op, loss, accuracy])
    duration = time.time() - start_time
    assert not np.isnan(loss_value), 'Model diverged with loss = NaN'
    if step % 50 == 0: # 每隔50步打印一次训练信息
        num_examples_per_step = FLAGS.batch_size
        examples_per_sec = num_examples_per_step / duration
        sec_per_batch = float(duration)
        format_str = ('%s: step %d, loss = %.4f, training accuracy =\
            %.4f (%.1f examples/sec; %.3f 'sec/batch)')
        print (format_str % (datetime.now(), step, loss_value, training_\
            acc, examples_per_sec, sec_per_batch))
    if step % 100 == 0: # 每100步写入一次数据
        summary_str = sess.run(summary_op)
        summary_writer.add_summary(summary_str, step)
    if step % 1000 == 0 or (step + 1) == FLAGS.max_steps:
        # 每1000步保存一次模型
        checkpoint_path = os.path.join(FLAGS.checkpoint_dir, 'model.ckpt')
        saver.save(sess, checkpoint_path, global_step=step)

def main(argv=None):
    cifar10.maybe_download_and_extract()
    if tf.gfile.Exists(FLAGS.train_dir):
        tf.gfile.DeleteRecursively(FLAGS.train_dir)
    tf.gfile.MakeDirs(FLAGS.train_dir)
    if not tf.gfile.Exists(FLAGS.checkpoint_dir):
        tf.gfile.MakeDirs(FLAGS.checkpoint_dir)
    train()

if __name__ == '__main__':
    tf.app.run()

```

4. cifar10_eval.py 的代码及说明

```

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from datetime import datetime
import math
import time
import numpy as np
import tensorflow as tf
import cifar10

```

```

FLAGS = tf.app.flags.FLAGS
tf.app.flags.DEFINE_string('eval_dir', './cifar10/eval', """Directory where to
write event logs.""")
tf.app.flags.DEFINE_string('eval_data', 'test', """Either 'test' or 'train\
eval'.""")
tf.app.flags.DEFINE_string('checkpoint_dir', './cifar10/16_VGG/checkpoint',
    """Directory where to read model checkpoints.""")
tf.app.flags.DEFINE_integer('eval_interval_secs', 60 * 5, """How often to run
the eval.""")
tf.app.flags.DEFINE_integer('num_examples', 10000, """Number of examples to
run.""")
tf.app.flags.DEFINE_boolean('run_once', True, """Whether to run eval only
once.""")

def eval_once(saver, summary_writer, top_k_op, summary_op): # 进行一次评测
    with tf.Session() as sess:
        ckpt = tf.train.get_checkpoint_state(FLAGS.checkpoint_dir)
        # 根据检查点找到最新模型的文件名
        if ckpt and ckpt.model_checkpoint_path:
            saver.restore(sess, ckpt.model_checkpoint_path) # 从检查点还原
            global_step = ckpt.model_checkpoint_path.split('/')[-1].split('-')\
                [-1] # 提取 global_step
        else:
            print('No checkpoint file found')
            return
        coord = tf.train.Coordinator() # 创建一个协调器，管理线程
        try:
            threads = []
            for qr in tf.get_collection(tf.GraphKeys.QUEUE_RUNNERS):
                threads.extend(qr.create_threads(sess, coord=coord, daemon=True,
                    start=True))

            num_iter = int(math.ceil(FLAGS.num_examples / FLAGS.batch_size))
            true_count = 0 # 统计准确预测的数目
            total_sample_count = num_iter * FLAGS.batch_size
            step = 0
            while step < num_iter and not coord.should_stop():
                predictions = sess.run([top_k_op])
                true_count += np.sum(predictions)
                step += 1

            precision = true_count / total_sample_count # 计算准确率
            print('%s: precision @ 1 = %.3f' % (datetime.now(), precision))
            # 打印准确率
            summary = tf.Summary()
            summary.ParseFromString(sess.run(summary_op))
            summary.value.add(tag='Precision @ 1', simple_value=precision)
            summary_writer.add_summary(summary, global_step)
        except Exception as e:
            coord.request_stop(e)
            coord.request_stop()
            coord.join(threads, stop_grace_period_secs=10)

def evaluate(): # 评测函数
    with tf.Graph().as_default() as g:
        eval_data = FLAGS.eval_data == 'test' # 得到 CIFAR-10 测试集的图像和标签

```

```

images, labels = cifar10.inputs(eval_data=eval_data)
images = tf.image.resize_images(images, [128, 128]) # 调整图像的大小
_, _, logits = cifar10.inference(images, training=False) # 构建预测计算图
top_k_op = tf.nn.in_top_k(logits, labels, 1)
variable_averages = tf.train.ExponentialMovingAverage(cifar10.MOVING_
AVERAGE_DECAY)
variables_to_restore = variable_averages.variables_to_restore()
saver = tf.train.Saver(variables_to_restore)
summary_op = tf.summary.merge_all() # 根据汇总集合构建汇总操作
summary_writer = tf.summary.FileWriter(FLAGS.eval_dir, g)
while True:
    eval_once(saver, summary_writer, top_k_op, summary_op)
    if FLAGS.run_once:
        break
    time.sleep(FLAGS.eval_interval_secs)
def main(argv=None):
    cifar10.maybe_download_and_extract()
    if tf.gfile.Exists(FLAGS.eval_dir):
        tf.gfile.DeleteRecursively(FLAGS.eval_dir)
    tf.gfile.MakeDirs(FLAGS.eval_dir)
    evaluate()
if __name__ == '__main__':
    tf.app.run()

```

6.1.3 VGGNet 的物体图像分类案例

本节描述一个利用 VGGNet-16 在 TensorFlow 框架下进行物体图像分类的案例，其中使用 CIFAR-10 数据集。可以根据表 1.2 提供的地址下载。如果读者不想单独下载，在程序运行时也会自动下载。

按照 6.1.2 节的 4 个程序 `cifar10_input.py`、`cifar10.py`、`cifar10_train.py` 和 `cifar10_eval.py`，利用 VGGNet-16 对 CIFAR-10 数据集进行图像分类，可参照图 6.1 的训练命令执行。训练完成时的最终结果如图 6.2 所示，迭代训练 50 000 次后，程序运行结束，训练损失函数值为 0.0138，训练准确率为 100%。测试命令窗口如图 6.3 所示，测试信息和最终测试结果如图 6.4 所示，测试准确率为 84.60%。

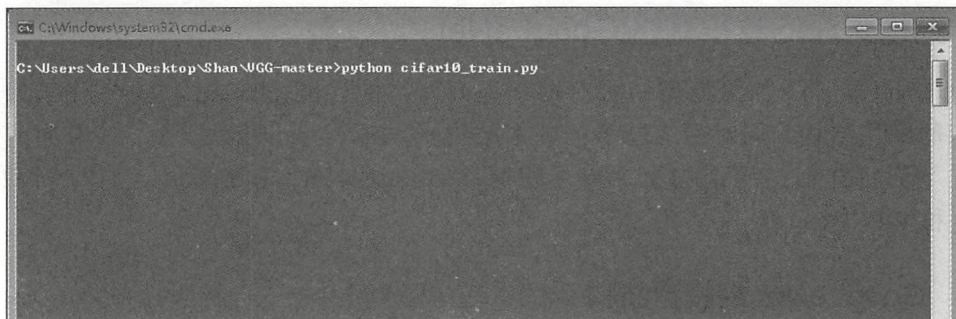


图 6.1 VGGNet 物体图像分类案例程序的训练运行命令


```

C:\Windows\system32\cmd.exe
2017-10-17 05:12:45.260258: step 48600, loss = 0.0127, training accuracy = 0.9922 (248.6 examples/sec; 0.51
2017-10-17 05:13:12.887907: step 48650, loss = 0.0094, training accuracy = 1.0000 (241.3 examples/sec; 0.53
2017-10-17 05:13:39.532754: step 48700, loss = 0.0110, training accuracy = 0.9922 (234.4 examples/sec; 0.54
2017-10-17 05:14:07.051202: step 48750, loss = 0.0122, training accuracy = 1.0000 (241.3 examples/sec; 0.53
2017-10-17 05:14:33.742849: step 48800, loss = 0.0330, training accuracy = 0.9922 (234.4 examples/sec; 0.54
2017-10-17 05:15:01.292497: step 48850, loss = 0.0479, training accuracy = 0.9844 (234.4 examples/sec; 0.54
2017-10-17 05:15:27.968544: step 48900, loss = 0.0492, training accuracy = 0.9844 (241.3 examples/sec; 0.53
2017-10-17 05:15:55.658593: step 48950, loss = 0.1112, training accuracy = 0.9844 (248.6 examples/sec; 0.51
2017-10-17 05:16:22.397040: step 49000, loss = 0.0247, training accuracy = 0.9922 (241.3 examples/sec; 0.53
2017-10-17 05:16:55.235098: step 49050, loss = 0.0499, training accuracy = 0.9688 (241.3 examples/sec; 0.53
2017-10-17 05:17:21.911144: step 49100, loss = 0.0652, training accuracy = 0.9766 (241.3 examples/sec; 0.53
2017-10-17 05:17:49.569993: step 49150, loss = 0.0268, training accuracy = 0.9844 (241.3 examples/sec; 0.53
2017-10-17 05:18:16.292840: step 49200, loss = 0.0260, training accuracy = 0.9922 (241.3 examples/sec; 0.53
2017-10-17 05:18:44.060889: step 49250, loss = 0.0507, training accuracy = 0.9844 (241.3 examples/sec; 0.53
2017-10-17 05:19:10.705735: step 49300, loss = 0.0249, training accuracy = 0.9922 (227.9 examples/sec; 0.56
2017-10-17 05:19:38.364584: step 49350, loss = 0.0141, training accuracy = 1.0000 (234.4 examples/sec; 0.54
2017-10-17 05:20:05.040631: step 49400, loss = 0.1179, training accuracy = 0.9609 (234.4 examples/sec; 0.54
2017-10-17 05:20:32.683879: step 49450, loss = 0.0091, training accuracy = 1.0000 (248.6 examples/sec; 0.51
2017-10-17 05:20:59.344326: step 49500, loss = 0.0369, training accuracy = 0.9844 (241.3 examples/sec; 0.53
2017-10-17 05:21:26.909575: step 49550, loss = 0.0113, training accuracy = 1.0000 (241.3 examples/sec; 0.53
2017-10-17 05:21:53.585622: step 49600, loss = 0.0067, training accuracy = 1.0000 (241.3 examples/sec; 0.53
2017-10-17 05:22:21.213270: step 49650, loss = 0.0411, training accuracy = 0.9766 (241.3 examples/sec; 0.53
2017-10-17 05:22:47.889317: step 49700, loss = 0.0129, training accuracy = 0.9922 (241.3 examples/sec; 0.53
2017-10-17 05:23:15.610566: step 49750, loss = 0.1312, training accuracy = 0.9609 (234.4 examples/sec; 0.54
2017-10-17 05:23:42.286612: step 49800, loss = 0.0359, training accuracy = 0.9922 (248.6 examples/sec; 0.51
2017-10-17 05:24:09.976661: step 49850, loss = 0.0455, training accuracy = 0.9844 (241.3 examples/sec; 0.53
2017-10-17 05:24:36.699508: step 49900, loss = 0.0151, training accuracy = 0.9922 (234.4 examples/sec; 0.54
2017-10-17 05:25:04.373957: step 49950, loss = 0.0138, training accuracy = 1.0000 (241.3 examples/sec; 0.53
C:\Users\dell\Desktop\Shan\UGG-master>

```

图 6.2 VGGNet 案例程序的训练结果

```

C:\Windows\system32\cmd.exe
C:\Users\dell\Desktop\Shan\UGG-master>python cifar10_eval.py

```

图 6.3 VGGNet 案例程序的测试命令

6.2 结构更深的卷积网络 GoogLeNet

VGGNet 验证了加深模型结构有助于提升网络的性能，GoogLeNet 专注于如何建立更深的网络结构，同时引入新型的基本结构——Inception 模块^[111]，以增加网络的宽度。下面将详细讨论结构更深的卷积网络 GoogLeNet，包括版本 V1^[112]、V2^[113]、V3^[114] 和 V4^[115]。

6.2.1 GoogLeNet 的模型结构

作为一种卷积网络的新模型，GoogLeNet 的关键在于使用了 Inception 模块。如图 6.5

所示, 每个原始 Inception 模块由前摄入层、并行处理层和过滤拼接层组成。并行处理层包括 4 个分支, 即 1×1 卷积分支、 3×3 卷积分支、 5×5 卷积分支和 3×3 最大池化分支。一个关于原始 Inception 模块的大问题是, 5×5 卷积分支即使采用中等规模的卷积核个数, 在计算代价上也可能是无法承受的。而且, 这个问题在混合池化之后会更为突出, 将导致输出单元不可避免地增加, 并很快出现计算量的暴涨。

```

C:\Windows\system32\cmd.exe
Total memory: 11.16GiB
Free memory: 11.09GiB
2017-10-17 09:31:36.897138: W c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\src\cuda\cuda_driver.cc:485] creating context when one is currently active; existing: 000000000765CE0
2017-10-17 09:31:37.287139: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\ine\gpu\gpu_device.cc:887] Found device 1 with properties:
name: Tesla K40c
major: 3 minor: 5 memoryClockRate (GHz) 0.745
pciBusID 0000:a1:00:0
Total memory: 11.16GiB
Free memory: 11.09GiB
2017-10-17 09:31:37.302739: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\ine\gpu\gpu_device.cc:779] Peer access not supported between device ordinals 0 and 1
2017-10-17 09:31:37.302739: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\ine\gpu\gpu_device.cc:779] Peer access not supported between device ordinals 1 and 0
2017-10-17 09:31:37.302739: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\ine\gpu\gpu_device.cc:988] DMA: 0 1
2017-10-17 09:31:37.302739: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\ine\gpu\gpu_device.cc:918] 0: Y N
2017-10-17 09:31:37.302739: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\ine\gpu\gpu_device.cc:918] 1: N Y
2017-10-17 09:31:37.302739: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\ine\gpu\gpu_device.cc:977] Creating TensorFlow device (/gpu:0) -> (device: 0, name: Tesla K40c, pci bus id:
)
2017-10-17 09:31:37.302739: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\ine\gpu\gpu_device.cc:977] Creating TensorFlow device (/gpu:1) -> (device: 1, name: Tesla K40c, pci bus id:
)
2017-10-17 09:31:51.483164: precision @ 1 = 0.846
C:\Users\dell\Desktop\Shan\UGG-master>

```

图 6.4 VGGNet 案例程序的最终测试结果

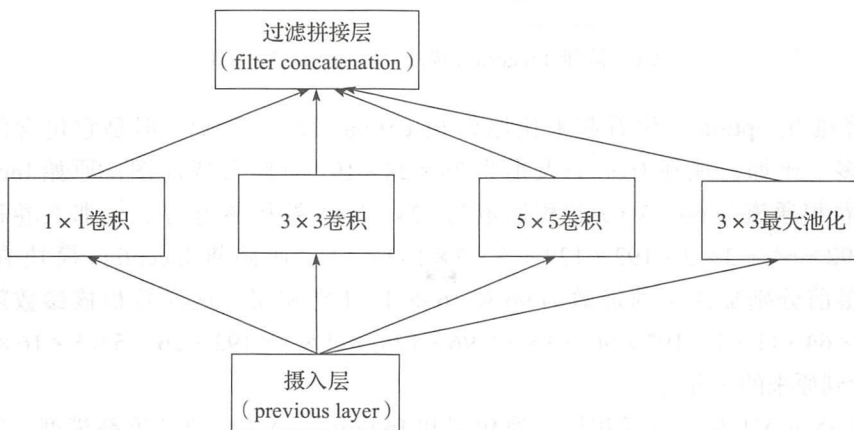


图 6.5 原始 Inception 模块

为了克服原始 Inception 模块的上述困难, 一种可行的策略是利用 1×1 卷积进行降维,

由此得到降维 Inception 模块，亦称 Inception V1 模块，如图 6.6 所示。与原始 Inception 模块类似，降维 Inception 模块也包括前摄入层、并行处理层和过滤拼接层。它们的不同之处在于并行处理层。降维 Inception 模块的并行处理层除了 1×1 卷积分支与原始 Inception 模块相同，其余 3 个分支都在原分支的基础上再串联了 1×1 卷积层。这 3 个分支分别命名为 $1 \times 1-3 \times 3$ 双卷积分支、 $1 \times 1-5 \times 5$ 双卷积分支、 $3 \times 3-1 \times 1$ 池化卷积分支。显然，降维 Inception 模块比原始 Inception 模块的结构更深，总体上有望获得更强的非线性表达能力。

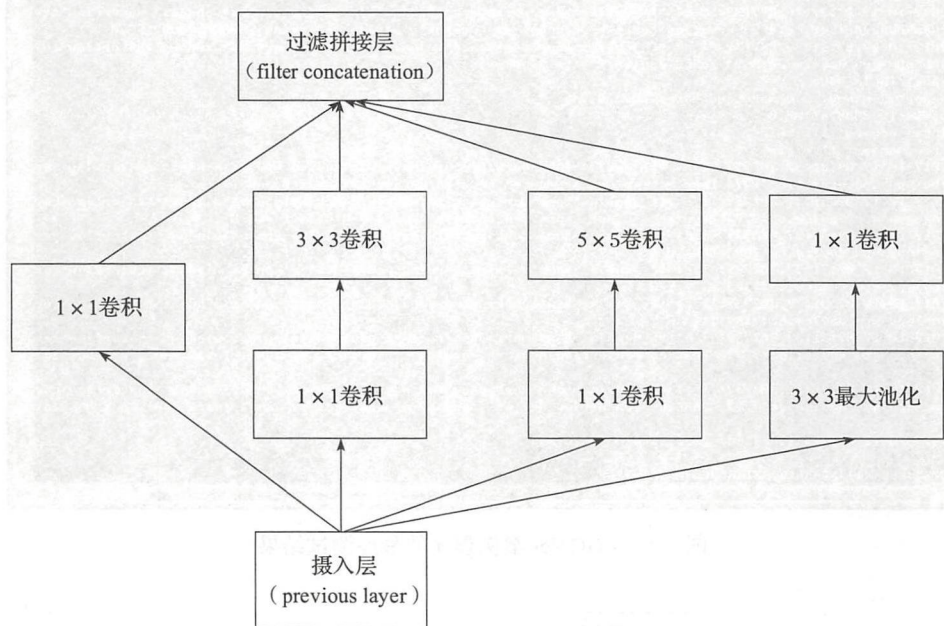


图 6.6 降维 Inception 模块 (Inception V1 模块)

虽然降维 Inception 模块看起来比原始 Inception 模块更复杂，但是它包含的参数往往要少得多。比如，现在有一个大小为 $28 \times 28 \times 192$ 的输入特征图，原始 Inception 模块中 1×1 卷积通道为 64， 3×3 卷积通道为 128， 5×5 卷积通道为 32，那么卷积核参数为 $1 \times 1 \times 192 \times 64 + 3 \times 3 \times 192 \times 128 + 5 \times 5 \times 192 \times 32$ ；而降维 Inception 模块在 3×3 和 5×5 卷积层前分别加入了通道数为 96 和 16 的 1×1 卷积层，这样卷积核参数就变成了 $1 \times 1 \times 192 \times 64 + (1 \times 1 \times 192 \times 96 + 3 \times 3 \times 96 \times 128) + (1 \times 1 \times 192 \times 16 + 5 \times 5 \times 16 \times 32)$ ，参数大约减少到原来的三分之一。

GoogLeNet V1 是一种卷积层、池化层和 Inception V1 模块的堆叠模型，结构如图 6.7 所示，参数如表 6.3 所示。不难看出，GoogLeNet V1 共包含 9 个 Inception V1 模块。其中，所有卷积层均采用 ReLU 激活函数，输入为减去均值后的 224×224 的彩色图像。



表 6.3 GoogLeNet V1 的参数

type	patch size/stride	output size	depth	#1 × 1	#3 × 3 reduce	#3 × 3	#5 × 5 reduce	#5 × 5	pool proj	params	ops
Conv	7 × 7/2	112 × 112 × 64	1							2.7K	34M
MaxPool	3 × 3/2	56 × 56 × 64	0								
Conv	3 × 3/1	28 × 28 × 192	2		64	192				112K	360M
MaxPool	3 × 3/2	28 × 28 × 192	0								
Inception V1 (3a)		28 × 28 × 256	2	64	96	128	16	32	32	159K	128M
Inception V1 (3b)		28 × 28 × 480	2	128	128	192	32	96	64	380K	304M
MaxPool	3 × 3/2	14 × 14 × 480	0								
Inception V1 (4a)		14 × 14 × 512	2	192	96	208	16	48	64	364K	73M
Inception V1 (4b)		14 × 14 × 512	2	160	112	224	24	64	64	437K	88M
Inception V1 (4c)		14 × 14 × 512	2	128	128	256	24	64	64	463K	100M
Inception V1 (4d)		14 × 14 × 528	2	112	144	288	32	64	64	580K	119M
Inception V1 (4e)		14 × 14 × 832	2	256	160	320	32	128	128	840K	170M
MaxPool	3 × 3/2	7 × 7 × 832	0								
Inception V1 (5a)		7 × 7 × 832	2	256	160	320	32	128	128	1 072K	54M
Inception V1 (5b)		7 × 7 × 1 024	2	284	192	384	48	128	128	1 388K	71M
AveragePool	7 × 7/1	1 × 1 × 1 024	0								
dropout (40%)		1 × 1 × 1 024	0								
Linear		1 × 1 × 1 000	1							1 000K	1M
Softmax		1 × 1 × 1 000	0								

在表 6.3 中，“#3 × 3 reduce”和“#5 × 5 reduce”表示因使用 1 × 1 卷积而减少的卷积核个数。pool proj 表示池化卷积分支中 max-pooling 之后的 1 × 1 卷积的个数；双卷积分支和池化卷积分支中的卷积层都要用 ReLU 激活函数。此外，分别在 Inception V1 (4a) 和 Inception V1 (4d) 上各增加一个辅助分类器，其结构分别如图 6.8 中从 Inception V1 (4a) 到 softmax0 的分支和从 Inception V1 (4d) 到 softmax1 的分支，其参数分别如表 6.4 和表 6.5 所示。在训练过程中，总损失由 softmax0、softmax1 和 softmax2 的加权构成，权重分别为 0.3、0.3 和 1.0。最终，GoogLeNet V1 在 ImageNet 上取得的 top-5 错误率为 6.67%。

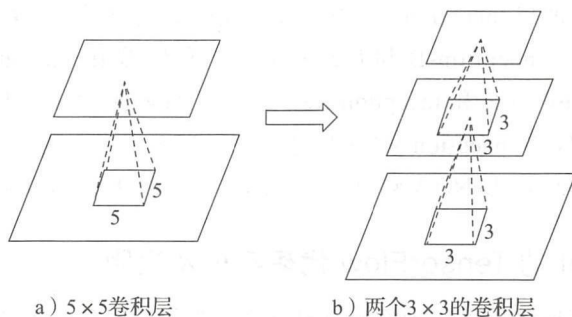
表 6.4 GoogLeNet V1 在模块 Inception V1(4a) 上的辅助分类器参数

type	patch size/stride	output size	depth
AveragePool	$5 \times 5/3$	$4 \times 4 \times 512$	512
Conv	$1 \times 1/1$	$4 \times 4 \times 128$	128
FC (ReLU)		$1 \times 1 \times 1024$	0
dropout (70%)		$1 \times 1 \times 1024$	0
FC (ReLU)		$1 \times 1 \times 1024$	0
FC		$1 \times 1 \times 1000$	0
Softmax		$1 \times 1 \times 1000$	0

表 6.5 GoogLeNet V1 在模块 Inception V1(4d) 上的辅助分类器参数

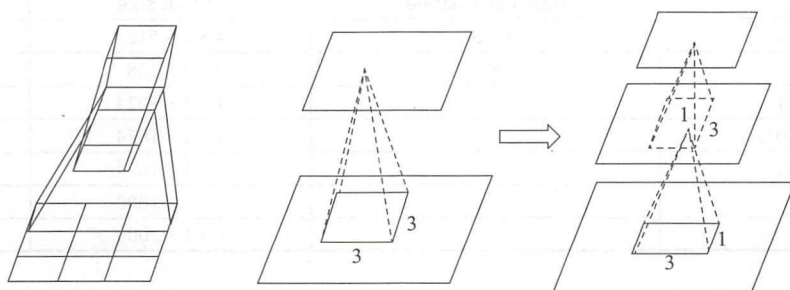
type	patch size/stride	output size	depth
AveragePool	$5 \times 5/3$	$4 \times 4 \times 528$	528
Conv	$1 \times 1/1$	$4 \times 4 \times 128$	128
FC (ReLU)		$1 \times 1 \times 1024$	0
dropout (70%)		$1 \times 1 \times 1024$	0
FC (ReLU)		$1 \times 1 \times 1024$	0
FC		$1 \times 1 \times 1000$	0
Softmax		$1 \times 1 \times 1000$	0

自 2014 年之后, Inception 模块不断改进, 现已发展到 V4。在 GoogLeNet V2 中, Inception V2 参考 VGGNet 用两个具有 3×3 核的卷积层代替具有 5×5 核的卷积层, 如图 6.8 所示。此外, GoogLeNet V2 减少了一个辅助分类器, 但同时引入了著名的 Batch Normalization(简称 BN)^[124]。BN 是一个非常有效的正则化方法, 可减少甚至取消 dropout 和 LRN 的使用, 同时加快训练速度、提高分类正确率。与 GoogLeNet V1 相比, GoogLeNet V2 的学习效率可以加快很多倍, 训练时间大大缩短, 在 ImageNet 上的 top-5 错误率可达 4.8%, 优于人类视觉水平。

图 6.8 Inception V2 把 Inception V1 的 5×5 卷积层替换成两个 3×3 的卷积层

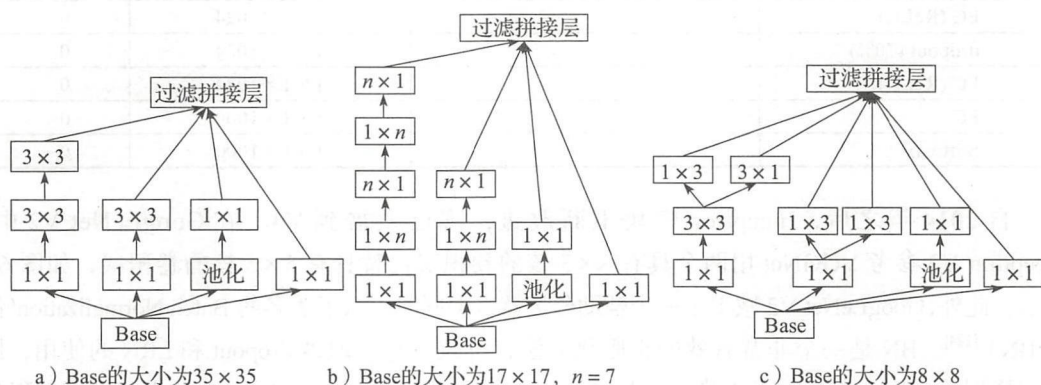
Inception V3 是通过改进 Inception V2 得到的^[125], 其中的核心思想是卷积分解, 也就是将一个较大的 $n \times n$ 二维卷积拆成两个较小的一维卷积 $n \times 1$ 和 $1 \times n$ 。比如, 将 3×3 卷积拆成 1×3 卷积和 3×1 卷积, 如图 6.9 所示。此外, Inception V3 有三种不同的结构(其 Base 的大小分别为 35×35 、 17×17 和 8×8), 如图 6.10 所示, 其中分支可能嵌套。与 GoogLeNet V2

类似，GoogLeNet V3 也只用了 1 个辅助分类器，在 ImageNet 上的最终 top-5 错误率为 3.5%。



a) 3×1 卷积核和 1×3 卷积核 b) 3×3 卷积层 c) 一个 3×1 层和一个 1×3 卷积层的堆叠结构

图 6.9 Inception V3 把 3×3 卷积核拆分成 3×1 卷积核和 1×3 卷积核，把 3×3 卷积层替换成一个 3×1 层和一个 1×3 卷积层的堆叠结构



a) Base 的大小为 35×35

b) Base 的大小为 17×17 , $n = 7$

c) Base 的大小为 8×8

图 6.10 Inception V3 的结构

Inception V4 是一种与 Inception V3 类似或更复杂的网络模块，可以分为纯粹 Inception V4（比如 Inception-A、Inception-B 和 Inception-C）和残差 Inception V4（比如 Inception-ResNet-A、Inception-ResNet-B 和 Inception-ResNet-C）。GoogLeNet V4 可以用纯粹 Inception V4 来构造，也可以用残差 Inception V4 来构造。有一种采用 1 个纯粹 Inception V4 和 3 个残差 Inception V4 构造的 GoogLeNet V4，在 ImageNet 上可以达到 3.08% 的 top-5 错误率^[126]。

6.2.2 GoogLeNet 的 TensorFlow 代码实现及说明

本节利用 TensorFlow 框架实现 GoogLeNet V3 的代码。GoogLeNet V3 的详细构造如图 6.11 所示。实现的 GoogLeNet V3 代码有 6 个程序，分别是 data_loader.py、arch.py、common.py、googlenet.py、train.py、eval.py。其中，data_loader.py 定义了网络的输入情况，包括将数据集做成队列形式，并重置大小等操作，为训练和测试做准备。arch.py、common.py 和 googlenet.py——这 3 个程序共同定义了网络的结构，其中 arch.py 定义了模型的加载情况，common.py 和 googlenet.py 共同定义了 GoogLeNet 网络的结构，包括网络训练和测

试时网络的结构。train.py 定义了网络的训练情况，包括训练数据的读入和存储路径、训练时各个参数大小、损失函数和训练过程中训练过程的打印显示等。eval.py 定义了网络的测试情况，包括测试数据的读入和存储路径、已训练好模型的载入和测试过程中训练情况的打印显示等。下面分别对这 6 个程序的代码进行详细说明。

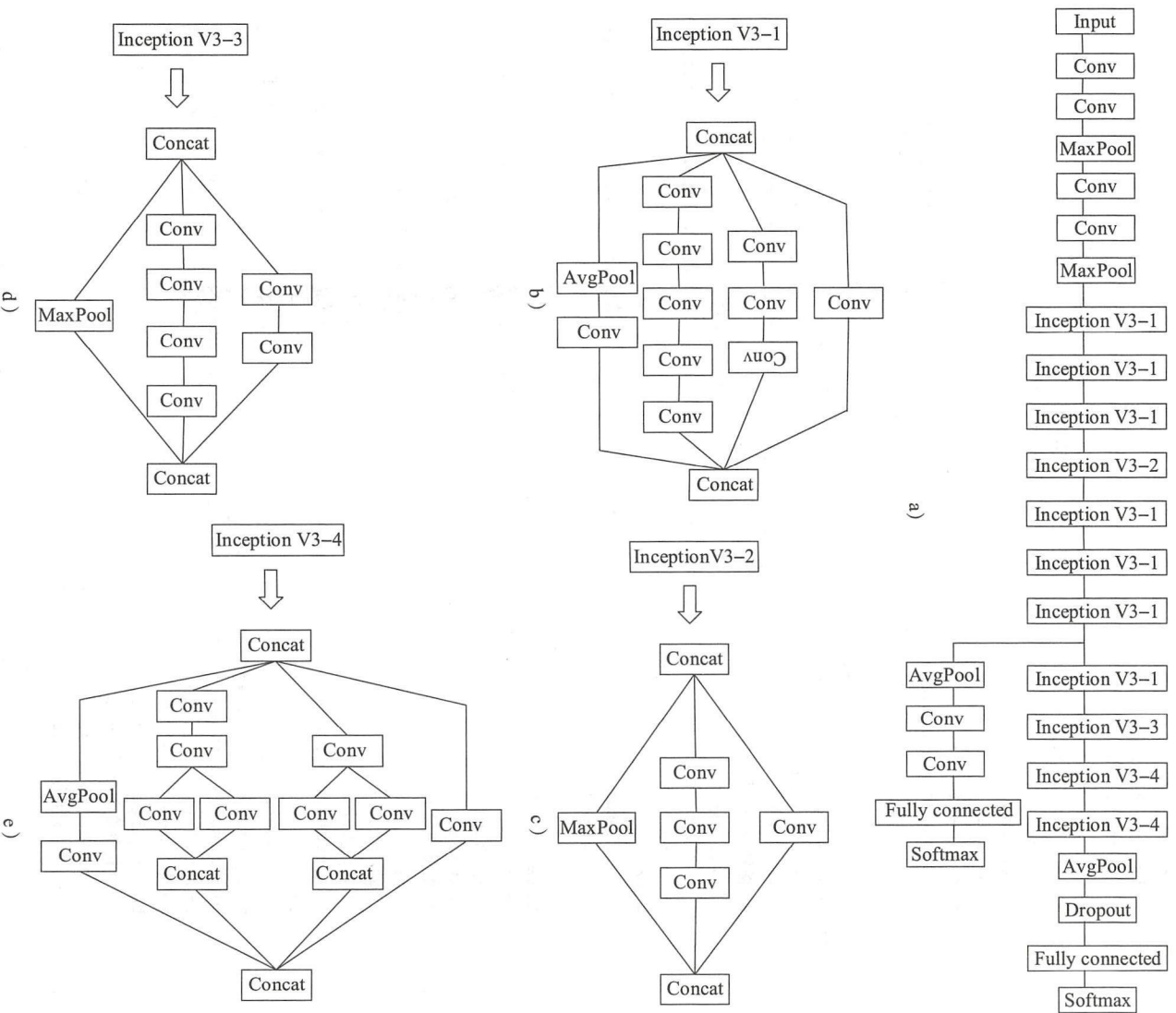


图 6.11 GoogleLeNet V3 示意图

1. data_loader.py 的代码及说明

```

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import os
import sys
from six.moves import xrange
import tensorflow as tf

def _read_label_file(file, delimiter): # 读入标签和文件
    f = open(file, "r") # 以读入形式打开文件
    filepaths = []
    labels = []
    for line in f:
        tokens = line.split(delimiter) # 以分隔符将 line 分为两部分并赋值给 tokens
        filepaths.append(tokens[0]) # 将 tokens[0] 添加到 filepaths 中
        labels.append(int(tokens[1])) # 将 int(tokens[1]) 添加到 labels 中
    return filepaths, labels

def read_inputs(is_training, args): # 读入数据函数
    filepaths, labels = _read_label_file(args.data_info, args.delimiter)
    filenames = [os.path.join(args.path_prefix, i) for i in filepaths]
    # 将路径和文件合并
    if is_training: # 创建一个文件读取的队列
        filename_queue = tf.train.slice_input_producer([filenames, labels], shuffle=\
args.shuffle, capacity= 1024)
    else:
        filename_queue = tf.train.slice_input_producer([filenames, labels], shuffle=\
False,
                                                    capacity= 1024, num_epochs =1)
    file_content = tf.read_file(filename_queue[0]) # 读取队列
    reshaped_image = tf.to_float(tf.image.decode_jpeg(file_content, channels=args.\
num_channels))
    reshaped_image = tf.image.resize_images(reshaped_image, args.load_size)
    # 重置图像大小
    label = tf.cast(filename_queue[1], tf.int64) # 转换数据类型
    img_info = filename_queue[0]
    if is_training:
        reshaped_image = _train_preprocess(reshaped_image, args) # 训练预处理
    else:
        reshaped_image = _test_preprocess(reshaped_image, args) # 测试预处理
    min_fraction_of_examples_in_queue = 0.4
    min_queue_examples = int(5000*min_fraction_of_examples_in_queue)
    print ('Filling queue with %d images before starting to train.' 'This may
take some times.' % min_queue_examples)
    batch_size = args.chunked_batch_size if is_training else args.batch_size
    # 加载图像和标签的附加信息
    if hasattr(args, 'save_predictions') and args.save_predictions is not
None: # 判断对象的 name 属性或方法
        images, label_batch, info = tf.train.batch( [reshaped_image, label,
img_info], batch_size= batch_size, num_threads=args.num_threads,
capacity=min_queue_examples+3*batch_size,

```



```

        allow_smaller_final_batch=True if not is_training else False)
    return images, label_batch, info
else:
    images, label_batch = tf.train.batch([reshaped_image, label], batch_
size= batch_size, allow_smaller_final_batch= True if not is_training
else False, num_threads=args.num_threads, capacity=min_queue_examples+3\
* batch_size)
    return images, label_batch

def _train_preprocess(reshaped_image, args):      # 定义训练时的预处理函数
    reshaped_image = tf.random_crop(reshaped_image, [args.crop_size[0], args.\
crop_size[1], args.num_channels])
    reshaped_image = tf.image.random_flip_left_right(reshaped_image)
    reshaped_image = tf.image.random_brightness(reshaped_image, max_delta=63)
    # 随机改变图片的亮度
    reshaped_image = tf.image.random_contrast(reshaped_image, lower=0.2, upper=1.8)
    # 随机改变色彩对比
    reshaped_image = tf.image.per_image_standardization(reshaped_image)
    # 减均值除方差标准化
    reshaped_image.set_shape([args.crop_size[0], args.crop_size[1], args.num\
channels])    # 设置张量大小
    return reshaped_image

def _test_preprocess(reshaped_image, args):      # 定义测试时的预处理函数
    resized_image=tf.image.resize_image_with_crop_or_pad(reshaped_image,args.\
crop_size[0], args.crop_size[1])
    float_image = tf.image.per_image_standardization(resized_image)
    float_image.set_shape([args.crop_size[0], args.crop_size[1], args.num\
channels])
    return float_image

```

2. arch.py 的代码及说明

```

import architectures.googlenet
def get_model(inputs, wd, is_training, args, transferMode= False):
    # 加载 GoogLeNet 的结构
    if args.architecture=='googlenet':
        return architectures.googlenet.inference(inputs, args.num_classes, wd,
0.4 if is_training
else 1.0, is_training, transferMode)

```

3. common.py 的代码及说明

```

import tensorflow as tf
import re
from tensorflow.python.training import moving_averages
from tensorflow.python.ops import control_flow_ops
from math import sqrt

RESNET_VARIABLES = 'resnet_variables'

```

```

TOWER_NAME = 'Tower'

def _get_variable(name, shape, initializer, regularizer=None, dtype='float',
trainable=True): # 初始化所有变量
    collections = [tf.GraphKeys.GLOBAL_VARIABLES, RESNET_VARIABLES]
    # 存储数据流图变量
    with tf.device('/cpu:0'):
        var = tf.get_variable(name, shape=shape, initializer=initializer, dtype=\
dtype, regularizer=regularizer, collections=collections, trainable=\
trainable)
    return var

def batchNormalization(x, is_training=True, decay=0.9, epsilon=0.001):
# BN 归一化函数
    x_shape = x.get_shape()
    params_shape = x_shape[-1:]
    axis = list(range(len(x_shape) - 1))
    beta = _get_variable('beta', params_shape, initializer=tf.zeros_initializer)
    gamma = _get_variable('gamma', params_shape, initializer=tf.ones_initializer)
    moving_mean = _get_variable('moving_mean', params_shape, initializer=tf.\
zeros_initializer, trainable=False)
    moving_variance = _get_variable('moving_variance', params_shape,
initializer=tf.ones_initializer, trainable=False)

    if is_training:
        mean, variance = tf.nn.moments(x, axis)
        update_moving_mean = moving_averages.assign_moving_average(moving_mean, mean,
decay)
        update_moving_variance = moving_averages.assign_moving_average(moving_\
variance, variance, decay)
        tf.add_to_collection(tf.GraphKeys.UPDATE_OPS, update_moving_mean)
        tf.add_to_collection(tf.GraphKeys.UPDATE_OPS, update_moving_variance)
        return tf.nn.batch_normalization(x, mean, variance, beta, gamma, epsilon)
    else:
        return tf.nn.batch_normalization(x, moving_mean, moving_variance, beta,
gamma, epsilon)

def flatten(x): # 平铺函数, 把张量改成向量
    shape = x.get_shape().as_list()
    dim = 1
    for i in xrange(1, len(shape)):
        dim *= shape[i]
    return tf.reshape(x, [-1, dim])

def treshold(x, treshold): # x 大于阈值时保持不变, 否则置为 0
    return tf.cast(x > treshold, x.dtype) * x

def fullyConnected(x, num_units_out, wd=0.0, weight_initializer=None, bias_\
initializer=None): # 定义全连接层
    num_units_in = x.get_shape()[1]
    stddev = 1. / tf.sqrt(tf.cast(num_units_out, tf.float32))
    if weight_initializer is None:
        weight_initializer = tf.random_uniform_initializer(minval=-stddev,
maxval=stddev, dtype=tf.float32)
    if bias_initializer is None:
        bias_initializer = tf.random_uniform_initializer(minval=-stddev, maxval=\
stddev, dtype=tf.float32)

```

```

weights = _get_variable('weights', [num_units_in, num_units_out], weight_
initializer, tf.contrib.layers.l2_regularizer(wd))
biases = _get_variable('biases', [num_units_out], bias_initializer)
return tf.nn.xw_plus_b(x, weights, biases)
def spatialConvolution(x, ksize, stride, filters_out, wd=0.0, weight_
initializer=None, bias_initializer=None):
    filters_in = x.get_shape()[-1]
    stddev = 1. / tf.sqrt(tf.cast(filters_out, tf.float32))
    if weight_initializer is None:
        weight_initializer = tf.random_uniform_initializer(minval=-stddev,
maxval=stddev, dtype=tf.float32)
    if bias_initializer is None:
        bias_initializer = tf.random_uniform_initializer(minval=-stddev, maxval=
stddev, dtype=tf.float32)
    shape = [ksize, ksize, filters_in, filters_out]
    weights = _get_variable('weights', shape, weight_initializer, tf.contrib.
layers.l2_regularizer(wd))
    conv = tf.nn.conv2d(x, weights, [1, stride, stride, 1], padding='SAME')
    biases = _get_variable('biases', [filters_out], bias_initializer)
    return tf.nn.bias_add(conv, biases)
def maxPool(x, ksize, stride):      # 定义最大池化函数
    return tf.nn.max_pool(x, ksize=[1, ksize, ksize, 1], strides=[1, stride,
stride, 1], padding='SAME')
def avgPool(x, ksize, stride):      # 定义平均池化函数
    return tf.nn.avg_pool(x, ksize=[1, ksize, ksize, 1], strides=[1, stride,
stride, 1], padding='SAME')

```

4. googlenet.py 的代码及说明

```

import tensorflow as tf
import common

def inception(x, conv1_size, conv3_size, conv5_size, pool1_size, wd, is_
training):      # 定义 Inception 模块
    with tf.variable_scope("conv_1"):
        conv1 = common.spatialConvolution(x, 1, 1, conv1_size, wd= wd)
        # 卷积运算
        conv1 = common.batchNormalization(conv1, is_training= is_training)
        # BN 归一化
        conv1 = tf.nn.relu(conv1)          # 使用 ReLU 激活函数
    with tf.variable_scope("conv_3_1"):
        conv3 = common.spatialConvolution(x, 1, 1, conv3_size[0], wd= wd)
        conv3 = common.batchNormalization(conv3, is_training= is_training)
        conv3 = tf.nn.relu(conv3)
    with tf.variable_scope("conv_3_2"):
        conv3 = common.spatialConvolution(conv3, 3, 1, conv3_size[1], wd= wd)
        conv3 = common.batchNormalization(conv3, is_training= is_training)
        conv3 = tf.nn.relu(conv3)
    with tf.variable_scope("conv_5_1"):
        conv5 = common.spatialConvolution(x, 1, 1, conv5_size[0], wd= wd)
        conv5 = common.batchNormalization(conv5, is_training= is_training)

```



```

conv5 = tf.nn.relu(conv5)
with tf.variable_scope("conv_5_2"):
    conv5 = common.spatialConvolution(conv5, 5, 1, conv5_size[1], wd= wd)
    conv5 = common.batchNormalization(conv5, is_training= is_training)
    conv5 = tf.nn.relu(conv5)
with tf.variable_scope("pool_1"):
    pool1= common.maxPool(x, 3, 1)
    pool1 = common.spatialConvolution(pool1, 1, 1, pool1_size, wd= wd)
    pool1 = common.batchNormalization(pool1, is_training= is_training)
    pool1 = tf.nn.relu(pool1)
return tf.concat([conv1, conv3, conv5, pool1], 3)      # 进行拼接并返回
def inference(x, num_output, wd, dropout_rate, is_training, transfer_mode=\
False): # 定义 GoogLeNet 的结构
    with tf.variable_scope('features'):                # 主分类器和辅助分类器的公共结构
        with tf.variable_scope('conv1'):
            network = common.spatialConvolution(x, 7, 2, 64, wd= wd)
            network = common.batchNormalization(network, is_training= is_training)
            network = tf.nn.relu (network)
        network = common.maxPool(network, 3, 2)
        with tf.variable_scope('conv2'):
            network = common.spatialConvolution(network, 1, 1, 64, wd= wd)
            network = common.batchNormalization(network, is_training= is_training)
            network = tf.nn.relu(network)
        with tf.variable_scope('conv3'):
            network = common.spatialConvolution(network, 3, 1, 192, wd= wd)
            network = common.batchNormalization(network, is_training= is_training)
            network = tf.nn.relu(network)
        network = common.maxPool(network, 3, 2)
        with tf.variable_scope('inception3a'):
            network = inception( network, 64, [96, 128], [16, 32], 32, wd= wd, is_\
training= is_training)
        with tf.variable_scope('inception3b'):
            network = inception( network, 128, [128, 192], [32, 96], 64, wd= wd,\
is_training= is_training)
        network = common.maxPool(network, 3, 2)
        with tf.variable_scope('inception4a'):
            network = inception( network, 192, [96, 208], [16, 48], 64, wd= wd,\
is_training= is_training)
        with tf.variable_scope('inception4b'):
            network = inception( network, 160, [112, 224], [24, 64], 64, wd= wd,\
is_training= is_training)
        with tf.variable_scope('inception4c'):
            network = inception( network, 128, [128, 256], [24, 64], 64, wd= wd,\
is_training= is_training)
        with tf.variable_scope('inception4d'):
            network = inception( network, 112, [144, 288], [32, 64], 64, wd= wd,\
is_training= is_training)
        with tf.variable_scope('mainb'):                # 主分类器
            with tf.variable_scope('inception4e'):
                main_branch = inception( network, 256, [160, 320], [32, 128], 128,\
wd= wd, is_training= is_training)
            main_branch = common.maxPool(main_branch, 3, 2)

```

```

with tf.variable_scope('inception5a'):
    main_branch= inception(main_branch, 256, [160, 320], [32, 128],
        128, wd= wd, is_training= is_training)
with tf.variable_scope('inception5b'):
    main_branch= inception(main_branch, 384, [192, 384], [48, 128],
        128, wd= wd, is_training= is_training)
main_branch= common.avgPool(main_branch, 7, 1)
main_branch= common.flatten(main_branch)
main_branch= tf.nn.dropout(main_branch, dropout_rate)
if not transfer_mode:
    with tf.variable_scope('output'):
        main_branch= common.fullyConnected(main_branch, num_output, wd= wd)
else:
    with tf.variable_scope('transfer_output'):
        main_branch= common.fullyConnected(main_branch, num_output, wd= wd)
with tf.variable_scope('auxb'):
    # 辅助分类器
    aux_classifier= common.avgPool(network, 5, 3)
    with tf.variable_scope('conv1'):
        aux_classifier= common.spatialConvolution(aux_classifier, 1, 1, 128,
            wd= wd)
        aux_classifier= common.batchNormalization(aux_classifier, is_training=
            is_training)
        aux_classifier= tf.nn.relu(aux_classifier)
    aux_classifier= common.flatten(aux_classifier)
    with tf.variable_scope('fcl'):
        aux_classifier= common.fullyConnected(aux_classifier, 1024, wd= wd)
        aux_classifier= tf.nn.dropout(aux_classifier, dropout_rate)
    if not transfer_mode:
        with tf.variable_scope('output'):
            aux_classifier= common.fullyConnected(aux_classifier, num_output,
                wd= wd)
    else:
        with tf.variable_scope('transfer_output'):
            aux_classifier= common.fullyConnected(aux_classifier, num_output,
                wd= wd)
return tf.concat([main_branch, aux_classifier],1) # 将主分支和辅助分类器拼接

```

5. train.py 的代码及说明

```

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from datetime import datetime
import os.path
import time
import numpy as np
from six.moves import xrange
import tensorflow as tf
import data_loader
import arch
import sys
import argparse

```

```

def loss(logits, labels):
    # 定义损失函数
    cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels, logits=logits,
                                                                    name='cross_entropy_per_example')
    cross_entropy_mean = tf.reduce_mean(cross_entropy, name='cross_entropy')
    tf.summary.scalar('Cross Entropy Loss', cross_entropy_mean) # 数据的汇总和记录
    return cross_entropy_mean

def average_gradients(tower_grads):
    # 定义平均梯度函数
    average_grads = []
    for grad_and_vars in zip(*tower_grads):
        # zip 函数可接受任意多个序列为参数，
        # 返回 tuple 列表

        grads = []
        for g, _ in grad_and_vars:
            expanded_g = tf.expand_dims(g, 0) # 扩展维度
            grads.append(expanded_g)
        grad = tf.concat(axis=0, values=grads)
        grad = tf.reduce_mean(grad, 0)
        v = grad_and_vars[0][1]
        grad_and_var = (grad, v)
        average_grads.append(grad_and_var)
    return average_grads

def train(args):
    # 定义训练过程
    with tf.Graph().as_default(), tf.device('/cpu:0'):
        images, labels = data_loader.read_inputs(True, args)
        epoch_number = tf.get_variable('epoch_number', [], dtype=tf.int32,
                                       initializer=tf.constant_initializer(0), trainable=False)
        lr = tf.train.piecewise_constant(epoch_number, [19, 30, 44, 53],
                                         [0.01, 0.005, 0.001, 0.0005, 0.0001], name='LearningRate')
        wd = tf.train.piecewise_constant(epoch_number, [30], [0.0005, 0.0],
                                         name='WeightDecay')
        opt = tf.train.MomentumOptimizer(lr, 0.9) # 使用动量优化方法
        tower_grads = []
        with tf.variable_scope(tf.get_variable_scope()):
            for i in xrange(args.num_gpus):
                with tf.device('/gpu:%d' % i):
                    with tf.name_scope('Tower_%d' % i) as scope:
                        logits = arch.get_model(images, wd, True, args)
                        toplacc = tf.reduce_mean(tf.cast(tf.nn.in_top_k(logits,
                                                                        labels, 1), tf.float32))
                        # top-1 准确率
                        top5acc = tf.reduce_mean(tf.cast(tf.nn.in_top_k(logits,
                                                                        labels, 5), tf.float32))
                        # top-5 准确率
                        cross_entropy_mean = loss(logits, labels)
                        regularization_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
                        reg_loss = tf.add_n(regularization_losses)
                        # 对应位置元素相加
                        tf.summary.scalar('Regularization Loss', reg_loss)
                        # 对 reg_loss 标量汇总和记录
                        total_loss = tf.add(cross_entropy_mean, reg_loss)
                        tf.summary.scalar('Total Loss', total_loss)
                        # 对 total_loss 标量汇总和记录
                        tf.summary.scalar('Top-1 Accuracy', toplacc)
                        # 对 toplacc 标量汇总和记录

```



```

        tf.summary.scalar('Top-5 Accuracy', top5acc)
        # 对top5acc 标量汇总和记录
        tf.get_variable_scope().reuse_variables()
        # 表示允许重用当前 scope 下所有变量
        summaries = tf.get_collection(tf.GraphKeys.SUMMARIES, scope)
        batchnorm_updates = tf.get_collection(tf.GraphKeys.
        UPDATE_OPS, scope)
        grads = opt.compute_gradients(total_loss)
        # 按批计算数据的梯度
        tower_grads.append(grads)
    grads = average_gradients(tower_grads)
    summaries.append(tf.summary.scalar('learning_rate', lr))
    summaries.append(tf.summary.scalar('weight_decay', wd))
    apply_gradient_op = opt.apply_gradients(grads)
    batchnorm_updates_op = tf.group(*batchnorm_updates)
    train_op = tf.group(apply_gradient_op, batchnorm_updates_op)
    saver = tf.train.Saver(tf.global_variables(), max_to_keep= args.num_epochs)
    summary_op = tf.summary.merge_all()
    init = tf.global_variables_initializer()
    if args.log_debug_info:
        run_options = tf.RunOptions(trace_level= tf.RunOptions.FULL_TRACE)
        run_metadata = tf.RunMetadata()
    else:
        run_options = None
        run_metadata = None
    sess = tf.Session(config=tf.ConfigProto(allow_soft_placement= True,
        log_device_placement= args.log_device_placement))
    if args.retrain_from is not None:
        saver.restore(sess, args.retrain_from)
    else:
        sess.run(init)
    tf.train.start_queue_runners(sess= sess)
    summary_writer = tf.summary.FileWriter(args.log_dir, sess.graph)
    start_epoch = sess.run(epoch_number + 1)
    for epoch in xrange(start_epoch, start_epoch + args.num_epochs):
        sess.run(epoch_number.assign(epoch))
        for step in xrange(args.num_batches):
            start_time = time.time()
            _, loss_value, top1_accuracy, top5_accuracy = sess.run( [train_op, cross_\
            entropy_mean, toplacc, top5acc], options= run_options, run_metadata=
            run_metadata)
            duration = time.time() - start_time
            assert not np.isnan(loss_value), 'Model diverged with loss = NaN'
            if step % 10 == 0:
                num_examples_per_step = args.chunked_batch_size * args.num_gpus
                examples_per_sec = num_examples_per_step / duration
                sec_per_batch = duration / args.num_gpus
                format_str = ('%s: epoch %d, step %d, loss = %.2f, Top-1 = %.2f Top-\
                5 = %.2f (%.1f examples/sec; %.3f ' 'sec/batch)')
                print (format_str % (datetime.now(), epoch, step, loss_value, top1_\
                accuracy, top5_accuracy, examples_per_sec, sec_per_batch))
                sys.stdout.flush()

```

等到程序执行完毕在屏幕上一次性输出结果

```

    if step % 100 == 0:
        summary_str = sess.run(summary_op)
        summary_writer.add_summary(summary_str, args.num_batches * epoch + step)
        # 写入文件
        if args.log_debug_info:
            summary_writer.add_run_metadata(run_metadata, 'epoch%d step%d'
                                           % (epoch, step))
        checkpoint_path = os.path.join(args.log_dir, args.snapshot_prefix) # 组合多个路径
        saver.save(sess, checkpoint_path, global_step= epoch)
def main():
    # 定义训练主函数
    parser = argparse.ArgumentParser(description='Process Command-line Arguments')
    # 创建解析器
    parser.add_argument('--load_size', nargs= 2, default= [256,256], type= int,
        action= 'store',
        help= 'The width and height of images for loading from disk')
    # 添加命令行参数和选项
    parser.add_argument('--crop_size', nargs= 2, default= [224,224], type= int,
        action= 'store',
        help= 'The width and height of images after random cropping')
    parser.add_argument('--batch_size', default= 128, type= int, action= 'store',
        help= 'The training batch size')
    parser.add_argument('--num_classes', default= 17 , type=int, action='store',
        help= 'The number of classes')
    parser.add_argument('--num_channels', default= 3 , type= int, action= 'store',
        help= 'The number of channels in input images')
    parser.add_argument('--num_epochs', default= 55, type= int, action= 'store',
        help= 'The number of epochs')
    parser.add_argument('--path_prefix', default= './', action='store', help=\
        'the prefix address for images')
    parser.add_argument('--data_info', default= 'train.txt', action= 'store',
        help= 'Name of the file containing addresses and labels of training
        images')
    parser.add_argument('--shuffle', default= True, type= bool, action= 'store',
        help= 'Shuffle training data or not')
    parser.add_argument('--num_threads', default= 20, type= int, action='store',
        help= 'The number of threads for loading data')
    parser.add_argument('--log_dir', default= None, action= 'store',
        help= 'Path for saving Tensorboard info and checkpoints')
    parser.add_argument('--snapshot_prefix', default= 'snapshot', action= 'store',
        help= 'Prefix for checkpoint files')
    parser.add_argument('--architecture', default= 'googlenet', help= 'The DNN
        architecture')
    parser.add_argument('--depth', default= 40, type= int, action= 'store',
        help= 'The depth of ResNet architecture')
    parser.add_argument('--run_name',
        default= 'Run'+str(time.strftime("%d-%m-%Y-%H-%M-%S")), action= 'store',
        help= 'Name of the experiment')
    parser.add_argument('--num_gpus', default= 1, type= int, action= 'store',
        help= 'Number of GPUs')
    parser.add_argument('--log_device_placement', default= False, type= bool,
        help= 'Whether to log device placement or not')
    parser.add_argument('--delimiter', default= ' ', action= 'store', help=\

```

```

'Delimiter of the input files')
parser.add_argument('--retrain_from', default= None, action= 'store',
                    help= 'Continue Training from a snapshot file')
parser.add_argument('--log_debug_info', default= False, action= 'store',
                    help= 'Logging runtime and memory usage info')
parser.add_argument('--num_batches', default= -1, type= int, action= 'store',
                    help= 'The number of batches per epoch')
args = parser.parse_args() # 解析参数列表
args.chunked_batch_size = int(args.batch_size/args.num_gpus)
args.num_samples = sum(1 for line in open(args.data_info))
if args.num_batches== -1:
    args.num_batches= int(args.num_samples/args.batch_size)+1
if args.log_dir is None:
    args.log_dir= args.architecture+"_"+args.run_name
print(args)
print("Saving everything in "+args.log_dir)
if tf.gfile.Exists(args.log_dir):
    tf.gfile.DeleteRecursively(args.log_dir) # 递归删除目录下的所有文件
tf.gfile.MakeDirs(args.log_dir) # 在 args.log_dir 目录下创建文件夹
train(args)
if __name__ == '__main__':
    main()

```

6. eval.py 的代码及说明

```

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from datetime import datetime
import math
import time
import os
import numpy as np
import tensorflow as tf
import argparse
import arch
import data_loader
import sys

def evaluate(args): # 评价函数
    with tf.Graph().as_default() as g, tf.device('/cpu:0'): # 建立数据图结构
        if args.save_predictions is None: # 得到图像和对应的标签
            images, labels = data_loader.read_inputs(False, args)
        else:
            images, labels, urls = data_loader.read_inputs(False, args)
        with tf.device('/gpu:0'): # 在 GPU 上执行计算
            logits = arch.get_model(images, 0.0, False, args) # 计算网络预测结果
            top_1_op = tf.nn.in_top_k(logits, labels, 1) # 计算 top-1 的预测准确率
            top_5_op = tf.nn.in_top_k(logits, labels, 5) # 计算 top-5 的预测准确率
            if args.save_predictions is not None:
                top5 = tf.nn.top_k(tf.nn.softmax(logits), 5)

```



```

        top5ind= top5.indices
        top5val= top5.values
        saver = tf.train.Saver(tf.global_variables())
        summary_op = tf.summary.merge_all()
        summary_writer = tf.summary.FileWriter(args.log_dir, g)
    with tf.Session(config=tf.ConfigProto(allow_soft_placement=True)) as sess:
        sess.run(tf.global_variables_initializer())
        sess.run(tf.local_variables_initializer())
        ckpt = tf.train.get_checkpoint_state(args.log_dir) # 加载训练好的模型
        if ckpt and ckpt.model_checkpoint_path: # 加载最新模型
            saver.restore(sess, ckpt.model_checkpoint_path) # 从检测点还原
        else:
            return
        coord = tf.train.Coordinator() # 启动队列运行程序
        threads = tf.train.start_queue_runners(sess=sess, coord=coord)
        true_predictions_count = 0 # 统计正确预测的数量
        true_top5_predictions_count = 0
        step = 0
        predictions_format_str = ('%d,%s,%d,%s,%s\n')
        batch_format_str = ('Batch Number: %d, Top-1 Hit: %d, Top-5 Hit: %d, Top-1 Accuracy: %.3f, Top-5 Accuracy: %.3f')
        if args.save_predictions is not None:
            out_file = open(args.save_predictions, 'w')
        while step < args.num_batches and not coord.should_stop():
            if args.save_predictions is None:
                top1_predictions, top5_predictions = sess.run([top1_op, top5_op])
            else:
                top1_predictions, top5_predictions, urls_values, label_values,
                top5guesses,
                top5conf = sess.run([top1_op, top5_op, urls, labels,
                                     top5ind, top5val])
                for i in xrange(0, urls_values.shape[0]):
                    out_file.write(predictions_format_str%(step*args.batch_size+i+1,
                                                            urls_values[i], label_values[i],
                                                            '[' + ', '.join('%d' % item for item in top5guesses[i]) + ']',
                                                            '[' + ', '.join('%.4f' % item for item in top5conf[i]) + ']'))
                    out_file.flush()
                true_predictions_count += np.sum(top1_predictions)
                true_top5_predictions_count += np.sum(top5_predictions)
                print(batch_format_str%(step, true_predictions_count, true_top5_predictions_count,
                                         true_predictions_count / ((step + 1.0) * args.batch_size),
                                         true_top5_predictions_count / ((step + 1.0) * args.batch_size)))
                sys.stdout.flush()
                step += 1
        if args.save_predictions is not None:
            out_file.close()
        summary = tf.Summary()
        summary.ParseFromString(sess.run(summary_op))
        coord.request_stop()
        coord.join(threads)

def main(): # 定义评价主函数

```

```

parser = argparse.ArgumentParser(description='Process Command-line Arguments')
# 创建解析器
parser.add_argument('--load_size', nargs= 2, default= [256,256], type= int,
action= 'store', help= 'The width and height of images for loading from disk')
parser.add_argument('--crop_size', nargs= 2, default= [224,224], type= int,
action= 'store', help= 'The width and height of images after random cropping')
parser.add_argument('--batch_size', default= 100, type= int, action= 'store',
help= 'The testing batch size')
parser.add_argument('--num_classes', default= 17, type= int, action= 'store',
help= 'The number of classes')
parser.add_argument('--num_channels', default= 3, type= int, action= 'store',
help= 'The number of channels in input images')
parser.add_argument('--num_batches', default=-1, type= int, action= 'store',
help= 'The number of batches of data')
parser.add_argument('--path_prefix', default='./', action= 'store', help= \
'The prefix address for images')
parser.add_argument('--delimiter', default=' ', action= 'store', help= 'Delimiter
for the input files')
parser.add_argument('--data_info', default= 'val.txt', action= 'store',
help= 'File containing the addresses and labels of testing
images')
parser.add_argument('--num_threads', default= 20, type= int, action= 'store',
help= 'The number of threads for loading data')
parser.add_argument('--architecture', default= 'resnet', help= 'The DNN
architecture')
parser.add_argument('--depth', default= 50, type= int, help= 'The depth of
ResNet architecture')
parser.add_argument('--log_dir', default= None, action= 'store',
help= 'Path for saving Tensorboard info and checkpoints')
parser.add_argument('--save_predictions', default= None, action= 'store',
help= 'Save top-5 predictions of the networks along with their confidence
in the specified file')
args = parser.parse_args() # 解析参数列表
args.num_samples = sum(1 for line in open(args.data_info))
if args.num_batches== -1:
    if (args.num_samples%args.batch_size==0):
        args.num_batches= int(args.num_samples/args.batch_size)
    else:
        args.num_batches= int(args.num_samples/args.batch_size)+1
print(args)
evaluate(args)

if __name__ == '__main__':
    main()

```

6.2.3 GoogLeNet 的鲜花图像分类案例

本节描述一个利用 GoogLeNet 在 TensorFlow 框架下对鲜花图像进行分类的案例，其中用到的 Oxford-17 数据集可以根据表 1.2 提供的地址下载。下载后将 Oxford-17 放在 ./data/ 目录下（也可根据需要放置在其他文件夹下），把训练图像存放在 train 文件夹下，把测试图

像存放到 test 文件夹下，如图 6.12 和图 6.13 所示。然后，创建训练标签文件 train.txt 和测试标签文件 test.txt。train.txt 由训练数据图片名称加标签构成，test.txt 由测试数据图片名称加标签构成，如图 6.14 和图 6.15 所示。train.txt 和 test.txt 创建好之后，与训练集和测试集一同存放在 ./data 所在的目录下。

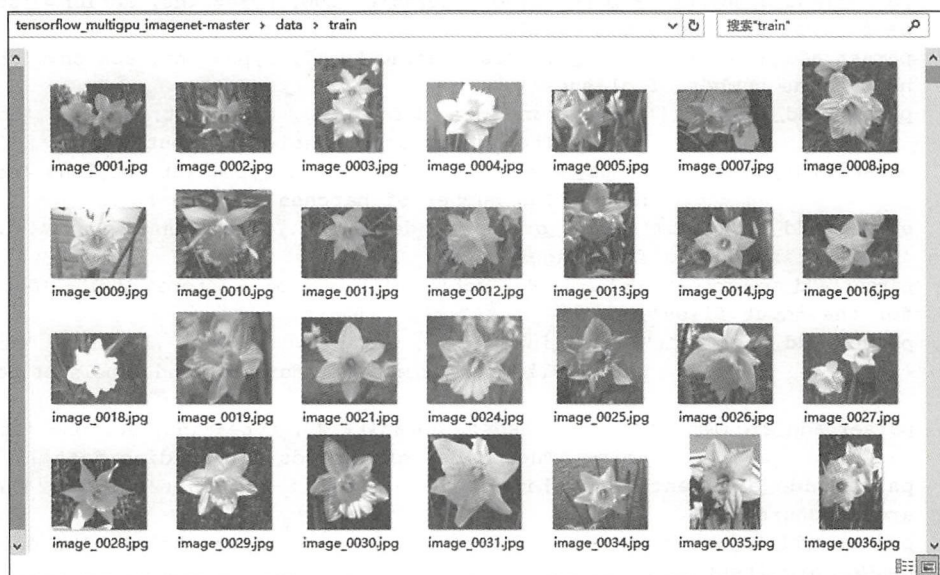


图 6.12 Oxford-17 鲜花图像的部分训练样本

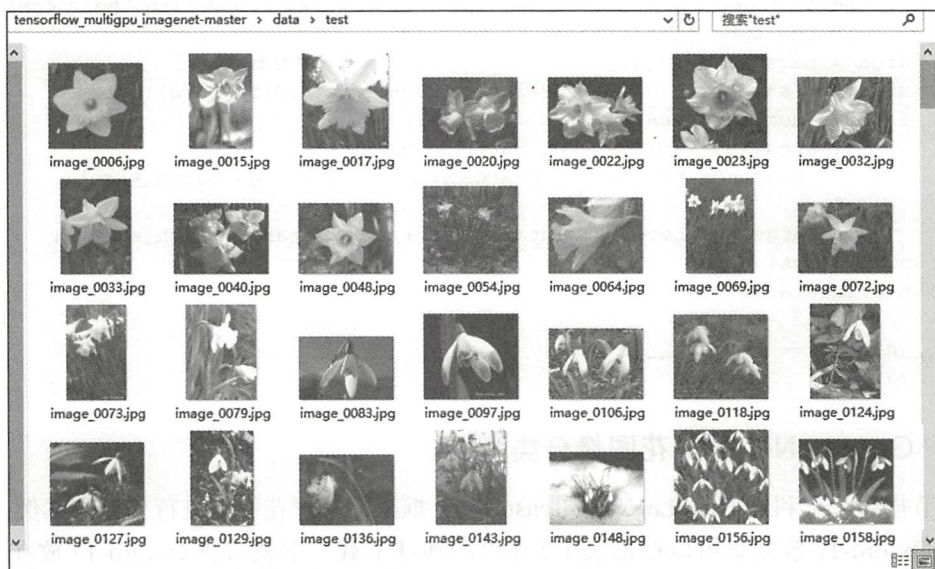


图 6.13 Oxford-17 鲜花图像的部分测试样本

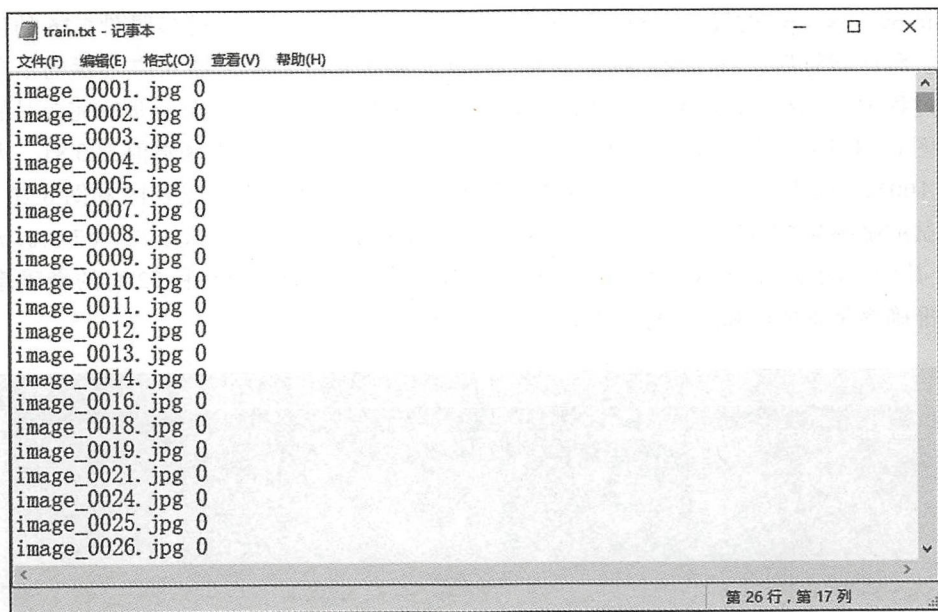


图 6.14 文件 train.txt 中的部分训练数据图片标签信息

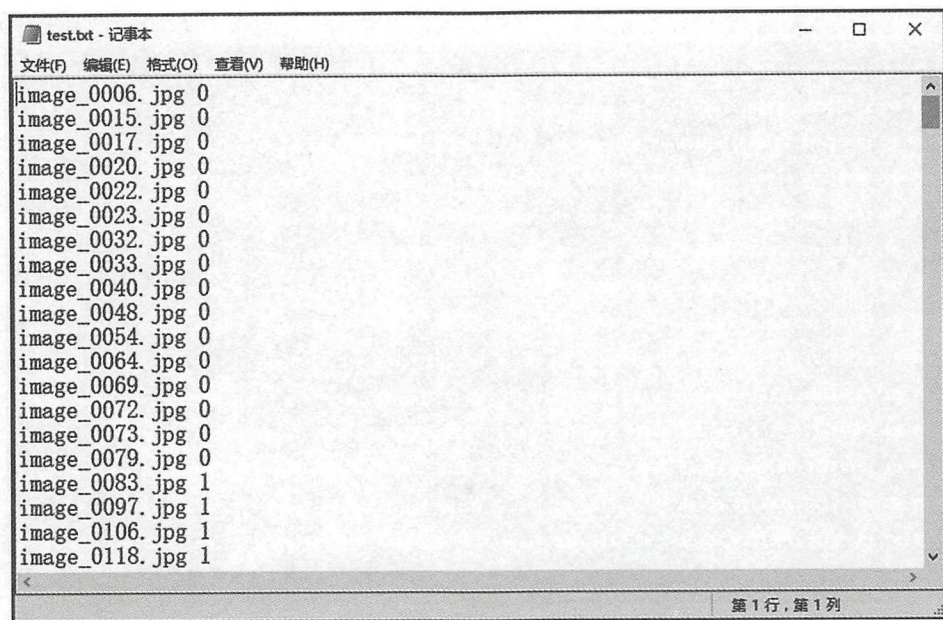


图 6.15 文件 test.txt 中的部分测试数据图片标签信息

在上述准备完成后, 基于 6.2.2 节程序 data_loader.py、arch.py、common.py、googlenet.

py、train.py、eval.py 的设置，利用 GoogLeNet 进行鲜花图像分类，可参照图 6.16 的训练命令运行。程序运行后，将出现一些信息窗口界面，如图 6.17 所示。图 6.17 显示的是在训练网络的过程中，随着迭代次数的增加，损失函数值和训练准确率的变化。训练完成时的最终结果如图 6.18 所示，迭代训练 5000 次后，程序运行结束，训练损失函数值为 0.81，训练准确率为 100%。训练集的命令窗口、测试信息和最终测试结果分别如图 6.19 和图 6.20 所示，平均测试准确率为 73.4%（其计算方法为： $(100 \times 78.0\% + 100 \times 73.5\% + 72 \times 67.0\%) / 272 = 0.734$ ，其中 78.0%、73.5% 和 67.0% 分别是迷你块大小为 100、100 和 72 时的准确率，所以总的准确率是总加权和除以总样本数，得出结果为 73.4%）。

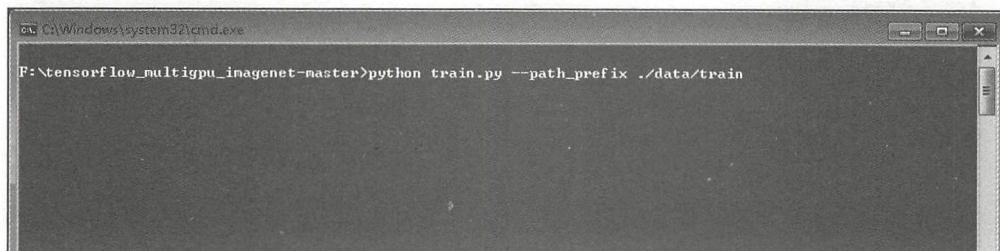


图 6.16 GoogLeNet 鲜花图像分类案例程序的训练命令

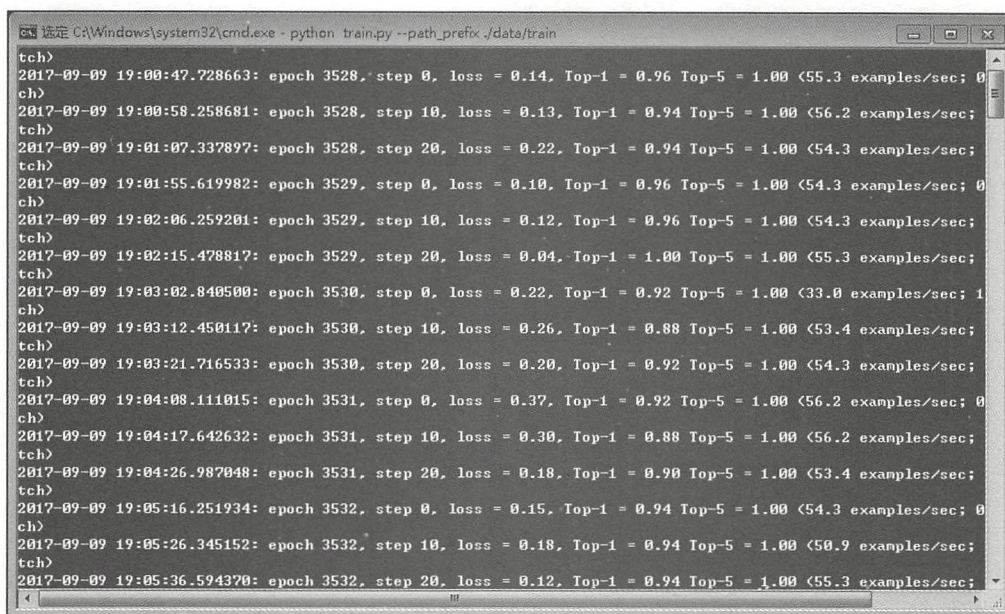


图 6.17 GoogLeNet 案例程序的训练信息


```

C:\Windows\system32\cmd.exe
ch>
2017-09-13 13:58:10.981209: epoch 4996, step 0, loss = 0.00, Top-1 = 1.00 Top-5 = 1.00 <51.7 examples/sec;
tch>
2017-09-13 13:58:20.700026: epoch 4996, step 20, loss = 0.08, Top-1 = 0.98 Top-5 = 1.00 <55.3 examples/sec;
tch>
2017-09-13 13:59:14.473321: epoch 4997, step 0, loss = 0.04, Top-1 = 1.00 Top-5 = 1.00 <54.3 examples/sec;
ch>
2017-09-13 13:59:24.379338: epoch 4997, step 10, loss = 0.06, Top-1 = 0.98 Top-5 = 1.00 <54.3 examples/sec;
tch>
2017-09-13 13:59:35.002957: epoch 4997, step 20, loss = 0.03, Top-1 = 0.98 Top-5 = 1.00 <55.3 examples/sec;
tch>
2017-09-13 14:00:27.029048: epoch 4998, step 0, loss = 0.01, Top-1 = 1.00 Top-5 = 1.00 <54.3 examples/sec;
ch>
2017-09-13 14:00:37.059866: epoch 4998, step 10, loss = 0.16, Top-1 = 0.96 Top-5 = 1.00 <53.4 examples/sec;
tch>
2017-09-13 14:00:47.449484: epoch 4998, step 20, loss = 0.02, Top-1 = 1.00 Top-5 = 1.00 <53.4 examples/sec;
tch>
2017-09-13 14:01:38.991975: epoch 4999, step 0, loss = 0.04, Top-1 = 0.98 Top-5 = 1.00 <55.3 examples/sec;
ch>
2017-09-13 14:01:49.038392: epoch 4999, step 10, loss = 0.02, Top-1 = 1.00 Top-5 = 1.00 <48.6 examples/sec;
tch>
2017-09-13 14:01:59.334410: epoch 4999, step 20, loss = 0.10, Top-1 = 0.98 Top-5 = 1.00 <51.7 examples/sec;
tch>
2017-09-13 14:02:50.767701: epoch 5000, step 0, loss = 0.13, Top-1 = 0.96 Top-5 = 1.00 <53.4 examples/sec;
ch>
2017-09-13 14:03:00.642518: epoch 5000, step 10, loss = 0.00, Top-1 = 1.00 Top-5 = 1.00 <56.2 examples/sec;
tch>
2017-09-13 14:03:11.000936: epoch 5000, step 20, loss = 0.01, Top-1 = 1.00 Top-5 = 1.00 <50.1 examples/sec;
tch>

```

图 6.18 GoogLeNet 案例程序的训练结果

```

C:\Windows\system32\cmd.exe
F:\tensorflow_multigpu_imagenet-master>python eval.py --num_threads 20 --architecture googlenet --log_dir "-
06-09-2017_10-26-46" --path_prefix ./data/test

```

图 6.19 GoogLeNet 案例程序的测试命令

```

C:\Windows\system32\cmd.exe
ine\gpu\gpu_device.cc:918] I:   N Y
2017-09-09 21:07:54.941138: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\cor
ine\gpu\gpu_device.cc:977] Creating TensorFlow device (/gpu:0) -> (device: 0, name: Tesla K40c, pci bus id:
)
2017-09-09 21:07:54.944138: I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\cor
ine\gpu\gpu_device.cc:977] Creating TensorFlow device (/gpu:1) -> (device: 1, name: Tesla K40c, pci bus id:
)
Batch Number: 0, Top-1 Hit: 78, Top-5 Hit: 95, Top-1 Accuracy: 0.780, Top-5 Accuracy: 0.950
Batch Number: 1, Top-1 Hit: 147, Top-5 Hit: 189, Top-1 Accuracy: 0.735, Top-5 Accuracy: 0.945
Batch Number: 2, Top-1 Hit: 201, Top-5 Hit: 254, Top-1 Accuracy: 0.670, Top-5 Accuracy: 0.847
F:\tensorflow_multigpu_imagenet-master>

```

图 6.20 GoogLeNet 案例程序的最终测试结果

卷积神经网络的跨连模型

标准卷积神经网络一般由卷积层、池化层和全连接层组成，其中每层只能与相邻层相连，也就是说，只能从相邻的前一层接收输入，并把输出传递给相邻的后一层。这种相邻层连接的网络结构不仅限制了卷积神经网络的多样性和灵活性，而且在结构加深时常常越来越难训练。一种有效的解决方案是引入跨层连接，建立卷积神经网络的跨连模型。跨连模型允许每层可以与非相邻层相连，既能从前面任意层接收输入，又能把其输出传递给后面的任意非相邻层。跨连卷积神经网络的深度可达到成百上千层，在大规模图像识别检测竞赛中获得过多项冠军。本章将介绍 4 种跨连模型，分别是快道网络、残差网络、密连网络和拼接网络，包括它们的模型结构，后三种的 Caffe 代码、应用案例及演示效果。

7.1 快道网络 HighwayNet

随着网络结构的不断加深，梯度消失或爆炸的问题会越来越严重，可能导致神经网络的学习和训练变得越来越困难。通过初始化、丢失输出、丢失连接和块归一化等技巧，这种困难能够得到一定程度的缓解。另一种解决办法是在网络中增加信息传递的快速通路，建立快道网络^[127]。在快道网络中，信息可以无障碍地跨越多层直接传递到后面的层。

在普通的分层神经网络中，每层都是对输入进行一个非线性映射变换，表示如下：

$$y = H(x, W_H) \quad (7.1)$$

其中， H 为非线性函数， W 表示权值矩阵， x 表示输入， y 表示输出。

快道网络的基本思想是在网络中定义两个非线性变换 $T(x, W_T)$ 和 $C(x, W_C)$ ，构造快道层：

$$y = H(x, W_H) \circ T(x, W_T) + x \circ C(x, W_C) \quad (7.2)$$

其中， T 表示变换门 (transform gate)， C 表示传递门 (carry gate)，符号 “ \circ ” 表示逐元素相乘。

令 $C = 1 - T$ ，快道层可简化为：

$$y = H(x, W_H) \circ T(x, W_T) + x \circ (1 - T(x, W_H)) \quad (7.3)$$

其中, x 、 y 、 $H(x, W_H)$ 和 $T(x, W_T)$ 必须有相同的维数。

不难看出, 如果选择特殊的变换门 T , 那么可以得到:

$$y = \begin{cases} x, & T(x, W_T) = 0 \\ H(x, W_H), & T(x, W_T) = 1 \end{cases} \quad (7.4)$$

因此, 根据变换门的输出, 快速层可以直接通过其输入 x 、直接输出 H , 或者在 x 和 H 之间产生平缓变化的输出。如果把快速层分解为若干块 (或单元), 用 $H_i(x)$ 表示第 i 个块的状态, 用 $T_i(x)$ 表示第 i 个变换门的输出, 那么快速层的第 i 个块的输出为

$$y_i = H_i(x) * T_i(x) + x_i * (1 - T_i(x)) \quad (7.5)$$

值得一提的是, 如果 x 、 y 、 $H(x, W_H)$ 和 $T(x, W_T)$ 的维数不一致, 则需要通过某种处理使其一致。比如: 先对 x 进行下采样或 0 填充后得到维数一致的 \hat{x} , 再用 \hat{x} 代替 x 。另一种处理是先改变普通层的维数, 再堆叠快速层。基于这两种处理, 在卷积神经网络中通过给定 H 和 T 使用共享权值及感受野的思想, 也可以构造类似的快速层。

在实际应用时, 可以把变换门定义为 $T(x) = \text{sigm}(W_T^T x + b_T)$, 其中 W_T 表示变换门的权值矩阵, b_T 表示变换门的偏置, $\text{sigm}(x) = \frac{1}{1 + e^{-x}}$ 。此时, 如果把 b_T 初始化为一个负值 (例如, -1 、 -3 等), 则可以使其在开始时偏向传递行为 (carry behavior)。

7.2 残差网络 ResNet

7.2.1 ResNet 的模型结构

随着层数的增加, 深度网络一般会越难训练。有些网络在开始收敛时, 还可能出现退化问题, 导致准确率很快达到饱和, 出现层次越深、错误率反而越高的现象。更令人意外的是, 这种退化导致的更高错误率并不是由于过拟合引起的, 而仅仅是因为增加了更多的层数^[128]。深度残差学习 (deep residual learning) 框架的提出^[68], 主要动机就是为了解决退化问题, 以便能够成功训练成百上千层的残差网络 (residual net)。

与普通神经网络的区别在于, 残差网络引入了跨层连接, 或者称为捷径连接 (shortcut connection), 构造了残差模块。如图 7.1 所示, 在一个残差模块中, 跨层连接一般只跨越 2~3 层, 但不排斥跨越更多的层。仅跨越 1 层的情况意义不大, 实验效果也不理想。如果从残差模块中去掉跨层连接, 并用 $H(x)$ 表示相应的计算结果, 那么在加入跨层连接后的计算结果

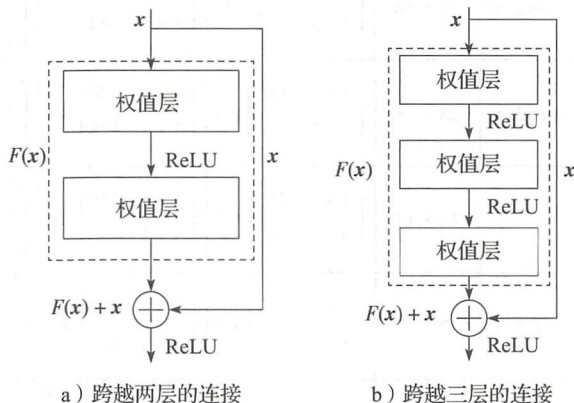


图 7.1 残差模块

$F(x)$ 与 $H(x)$ 之间存在下面的关系：

$$F(x) := H(x) - x \quad (7.6)$$

这意味着，跨连模块计算的是不跨连时的残差，所以又称为残差模块。从整体功能上看，如果用 $\{W_i\}$ 表示残差模块的所有权值，那么残差模块实际上计算的是下面的输出结果：

$$y = F(x, \{w_i\}) + x \quad (7.7)$$

其中， $F(x, \{w_i\})$ 又称为残差映射（或残差函数），是需要学习的。对于两个权值层的情况图 7.1a，在忽略偏置的情况下， $F(x, \{w_i\}) = w_2 \sigma(w_1 x) = w_2 \text{ReLU}(w_1 x)$ 。

残差模块的计算要求 $F(x, \{w_i\})$ 和 x 具有相同的维数。如果它们的维数不同，就需要引入一个额外的权值矩阵 w_s 对 x 进行线性投影，使得它们的维数相同，相应的计算结果为

$$y = F(x, \{w_i\}) + w_s x \quad (7.8)$$

既可以用全连接层构造残差模块，也可以用卷积层构造残差模块。基于残差模块，深层残差网络可以具有非常深的结构，深度甚至可达 1000 层以上。表 7.1 给出了 5 个残差网络结构的详细描述，包含它们的层数（分别是 18、34、50、101 和 152）、层名（layer name）、输出大小（output size）、残差模块（用方括号表示）等，其中有些层做过补 0 处理。在表 7.1 中，每个残差模块右边的数字代表串联重复的次数。比如，在 18 层的残差网络中，残差模块 conv3_x 串联重复两次。也就是说，网络中包含两个串联的 conv3_x。每个 conv3_x 都由两个卷积层和跨越它们的连接组成，其中两个卷积层分别是 conv3_1 $[3 \times 3, 128]$ 和 conv3_2 $[3 \times 3, 128]$ 。网络的复杂度是用浮点计算次数 FLOP 来度量的，比如对 18 层的网络， $\text{FLOP} = 1.8 \times 10^9$ 。

表 7.1 用于 ImageNet 的 5 种深层残差网络结构

层名	输出大小	18 层	34 层	50 层	101 层	152 层
conv1	112 × 112	7 × 7, 64, 步长 2				
maxpool	56 × 56	3 × 3 最大池化, 步长 2				
conv2_x	56 × 56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28 × 28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14 × 14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7 × 7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
avgpool	1 × 1	平均池化, 1000 维 fc, 软最大输出				
FLOP		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

另外,从表 7.1 可以看出,50 层的残差网络包括输入层 (image)、1 个独立卷积层 (conv1)、1 个最大池化层 (maxpool)、4 种卷积残差模块 (分别为 conv2_x、conv3_x、conv4_x 和 conv5_x)、1 个平均池化层 (avgpool) 和 1 个软最大输出层,详细结构如图 7.2 所示。具体来说,这个 50 层的残差网络,输入是大小为 $224 \times 224 \times 3$ 的三维数组,第 1 个卷积层是独立卷积层,使用 64 个大小为 7×7 、步长为 2 的卷积核,输出的大小为 112×112 。之后是一个最大池化层,池化窗口和步长分别为 3×3 和 2。接着是 4 种不同的卷积残差模块,3 个 conv2_x、4 个 conv3_x、6 个 conv4_x 和 3 个 conv5_x。每个卷积残差模块由 2~3 个卷积层和跨越它们的连接组成,比如,每个 conv2_x 类型的残差模块包含 conv2_1[1×1 , 64]、conv2_2[3×3 , 64]、conv2_3[1×1 , 256] 三个卷积层和跨越它们的连接。在这个 50 层残差网络的最后,是平均池化层和 1000 维的全连接软最大输出层,用于区分 1000 个不同类别。注意,网络的层数不统计池化层。

最后,在 ResNet 的基础上又产生了一些新的变种模型,比如 ResNeXt^[129]、特征金字塔网络 (Feature Pyramid Network, FPN)^[130]、宽度残差网络 (Wide Residual Network, WRN)^[131]。与残差网络相比,ResNeXt 增加了一个新的基数维度,基数就是并行重复残差模块的个数。FPN 同时使用多尺度特征代替单一尺度特征进行目标检测。WRN 则进一步在跨层连接中增加了需要训练的卷积层。

7.2.2 ResNet 的 Caffe 代码实现及说明

残差网络的 Caffe 实现代码的下载地址为 <https://github.com/KaimingHe/deep-residual-networks>。在这个地址的子文件夹 prototxt 下共包含 3 种残差网络的代码,分别为 50 层、101 层和 152 层的结构。其中,文件 ResNet-50-deploy.prototxt 只定义了 50 层网络的验证结构。下面参考表 7.1 对这个文件的几个关键模块进行说明,包括独立卷积层 conv1、最大池化层 maxpool、卷积残差模块 conv2_x、平均池化层 avgpool 和软最大输出层。

1. 独立卷积层 conv1 的实现代码及说明

```
layer {
  bottom: "data"           # 底层为输入数据
  top: "conv1"
  name: "conv1"
  type: "Convolution"      # 表示该层为卷积层
  convolution_param {
    num_output: 64          # 输出特征映射的个数
    kernel_size: 7          # 卷积核的大小
    pad: 3                  # 对输入的四边各扩展 3 个单位长度,并用 0 填充
    stride: 2               # 卷积核的步长
  }
}
layer {
  bottom: "conv1"
  top: "conv1"
```

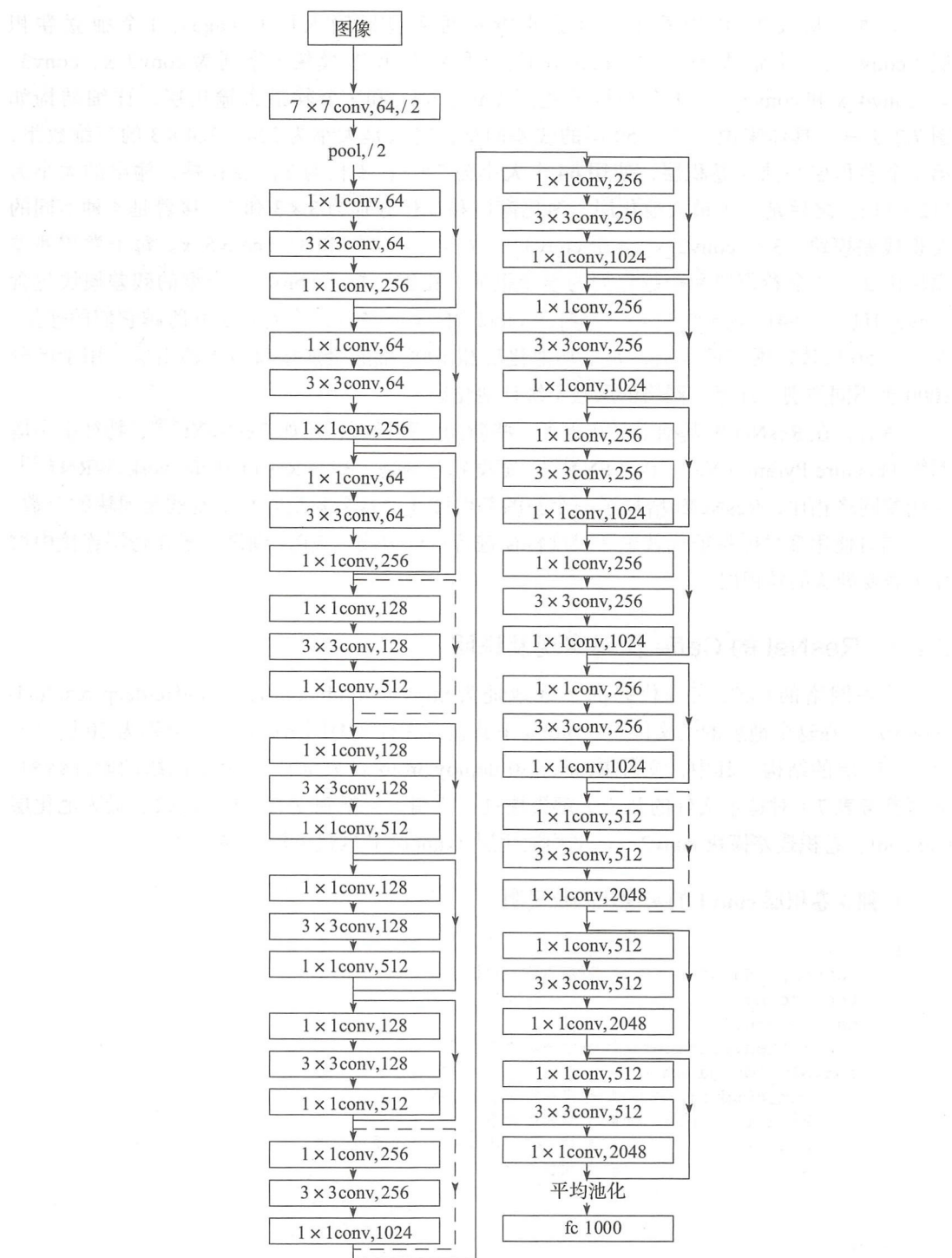


图 7.2 用于 ImageNet 分类的 50 层残差网络结构

```

    name: "bn_conv1"
    type: "BatchNorm"           # 表示该层是块归一化层
    batch_norm_param {
        use_global_stats: true  # 表示直接使用保存的均值和方差进行块归一化
    }
}
layer {
    bottom: "conv1"
    top: "conv1"
    name: "scale_conv1"
    type: "Scale"               # 表示尺度变换层
    scale_param {
        bias_term: true        # 表示在学习变换因子的同时，还学习偏置
    }
}
layer {
    bottom: "conv1"
    top: "conv1"
    name: "conv1_relu"
    type: "ReLU"               # 把激活函数选为 ReLU
}

```

2. 最大池化层 maxpool 的实现代码及说明

```

layer {
    bottom: "conv1"
    top: "pool1"
    name: "pool1"
    type: "Pooling"
    pooling_param {
        kernel_size: 3          # 池化窗口的大小
        stride: 2               # 池化窗口的步长
        pool: MAX               # 把池化方式选为最大池化
    }
}

```

3. 卷积残差模块 conv2_x 的实现代码及说明

```

# conv2_x 的跨层连接定义
layer {
    bottom: "pool1"            # 第 1 个跨层连接从最大池化层 maxpool 开始
    top: "res2a_branch1"
    name: "res2a_branch1"      # 把这个跨层连接本身看作一个维数变换层
    type: "Convolution"
    convolution_param {
        num_output: 256        # 变换后特征图的个数是 256
        kernel_size: 1
        pad: 0                 # 对输入的四边不进行扩展和填充
        stride: 1
        bias_term: false       # 不学习偏置
    }
}

```



```

layer {
  bottom: "res2a_branch1"
  top: "res2a_branch1"
  name: "bn2a_branch1"
  type: "BatchNorm"
  batch_norm_param {
    use_global_stats: true
  }
}
layer {
  bottom: "res2a_branch1"
  top: "res2a_branch1"
  name: "scale2a_branch1"
  type: "Scale"
  scale_param {
    bias_term: true
  }
}

# conv2_x 的第 1 个卷积层定义
layer {
  bottom: "pool1"
  top: "res2a_branch2a"
  name: "res2a_branch2a"
  type: "Convolution" # 表示该层是卷积层
  convolution_param {
    num_output: 64 # 卷积核的个数
    kernel_size: 1 # 卷积核的大小
    pad: 0 # 对输入的四边不进行扩展和填充
    stride: 1 # 卷积核的步长
    bias_term: false # 不学习偏置
  }
}
layer {
  bottom: "res2a_branch2a"
  top: "res2a_branch2a"
  name: "bn2a_branch2a"
  type: "BatchNorm"
  batch_norm_param {
    use_global_stats: true
  }
}
layer {
  bottom: "res2a_branch2a"
  top: "res2a_branch2a"
  name: "scale2a_branch2a"
  type: "Scale"
  scale_param {
    bias_term: true
  }
}
layer {
  bottom: "res2a_branch2a"
  top: "res2a_branch2a"

```

```
    name: "res2a_branch2a_relu"
    type: "ReLU" # 把激活函数选为 ReLU
}

# conv2_x 的第 2 个卷积层定义
layer {
    bottom: "res2a_branch2a"
    top: "res2a_branch2b"
    name: "res2a_branch2b"
    type: "Convolution"
    convolution_param {
        num_output: 64
        kernel_size: 3
        pad: 1
        stride: 1
        bias_term: false
    }
}

layer {
    bottom: "res2a_branch2b"
    top: "res2a_branch2b"
    name: "bn2a_branch2b"
    type: "BatchNorm"
    batch_norm_param {
        use_global_stats: true
    }
}

layer {
    bottom: "res2a_branch2b"
    top: "res2a_branch2b"
    name: "scale2a_branch2b"
    type: "Scale"
    scale_param {
        bias_term: true
    }
}

layer {
    bottom: "res2a_branch2b"
    top: "res2a_branch2b"
    name: "res2a_branch2b_relu"
    type: "ReLU"
}

# conv2_x 的第 3 个卷积层定义
layer {
    bottom: "res2a_branch2b"
    top: "res2a_branch2c"
    name: "res2a_branch2c"
    type: "Convolution"
    convolution_param {
        num_output: 256
        kernel_size: 1
        pad: 0
        stride: 1
    }
}
```

```

        bias_term: false
    }
}
layer {
    bottom: "res2a_branch2c"
    top: "res2a_branch2c"
    name: "bn2a_branch2c"
    type: "BatchNorm"
    batch_norm_param {
        use_global_stats: true
    }
}
layer {
    bottom: "res2a_branch2c"
    top: "res2a_branch2c"
    name: "scale2a_branch2c"
    type: "Scale"
    scale_param {
        bias_term: true
    }
}
layer {
    bottom: "res2a_branch1"          # 跨层连接的维数变换输出
    bottom: "res2a_branch2c"        # conv2_x 的第 3 个卷积层的输出
    top: "res2a"
    name: "res2a"
    type: "Eltwise"                 # 把两个输出逐元素相加，得到 conv2_x 的激活函数输入
}
layer {
    bottom: "res2a"
    top: "res2a"
    name: "res2a_relu"
    type: "ReLU"                   # 经过 ReLU 变换，得到 conv2_x 的最终输出
}

```

4. 平均池化层 avgpool 的实现代码及说明

```

layer {
    bottom: "res5c"
    top: "pool5"
    name: "pool5"
    type: "Pooling"
    pooling_param {
        kernel_size: 7          # 池化窗口的大小
        stride: 1               # 池化窗口的步长
        pool: AVE               # 把池化方式选为平均池化
    }
}

```

5. 软最大输出层的实现代码及说明

```

layer {

```



```

    bottom: "pool5"
    top: "fc1000"
    name: "fc1000"
    type: "InnerProduct"          # 表示该层为全连接层
    inner_product_param {
        num_output: 1000
    }
}
layer {
    bottom: "fc1000"
    top: "prob"
    name: "prob"
    type: "Softmax"              # 表示该层为软最大输出层
}

```

7.2.3 ResNet 的大规模图像分类案例

本节描述一个利用 ResNet 在 Caffe 框架下进行大规模图像分类的案例，其中用到的 ImageNet 数据集可以根据表 1.2 提供的地址下载。注意，这个数据集的图像是 .jpg 格式的，需要参考本书 3.6.1 节转换成 Caffe 支持的 LEVELDB 格式。此外，这个案例还需要在 ResNet-50-deploy.prototxt 的基础上编写网络结构文件 ResNet-50-train.prototxt 和 ResNet-50-validate.prototxt 分别用于训练和验证，并编写求解器配置文件 ResNet-50-solver.prototxt。下面分别进行描述。

1. 网络结构文件 ResNet-50-train.prototxt 的内容描述

这个文件主要是在 ResNet-50-deploy.prototxt 的基础上增加一些代码，包括增加训练集信息、增加卷积层的参数初始化方式、修改块归一化层的参数设置、增加网络损失计算层等。详细说明如下。

```

# 增加训练集信息
layer {
    name: "data"
    type: "Data"
    top: "data"
    top: "label"
    include {
        phase: TRAIN          # 表示用于训练的数据信息
    }
    transform_param {
        mirror: true          # 表示对图像进行水平翻转
        crop_size: 224        # 表示将图像裁剪为 224×224 大小
        mean_value: 104        # 表示 B 通道的均值
        mean_value: 117        # 表示 G 通道的均值
        mean_value: 123        # 表示 R 通道的均值
    }
    data_param {
        source: "F:/dataset/ilsvcr12/train_leveldb " # 表示训练集的存放路径
        batch_size: 8          # 表示训练集迷你块的大小
    }
}

```

```

        backend: LEVELDB          # 表示训练集的数据格式
    }
}

# 增加卷积层的参数初始化方式，例如卷积层 conv1 的完整定义如下
layer {
    bottom: "data"                # 底层为数据
    top: "conv1"
    name: "conv1"
    type: "Convolution"          # 表示该层是卷积层
    convolution_param {
        num_output: 64
        kernel_size: 7
        pad: 3
        stride: 2
        weight_filler {
            type: "msra"          # 使用 msra 方式初始化权值
        }
        bias_term: true          # 表示还要学习偏置
    }
}

# 修改块归一化层的参数设置
layer {
    bottom: "conv1"
    top: "conv1"
    name: "bn_conv1"
    type: "BatchNorm"            # 表示该层是块归一化层
    batch_norm_param {
        use_global_stats: false # 将此参数设置为 false，表示使用当前迷你块计算参数
    }
}

# 增加网络损失计算层
layer {
    bottom: "fc1000"              # 计算损失要用到全连接层 fc1000
    bottom: "label"              # 计算损失还要用到标签层
    name: "loss"
    type: "SoftmaxWithLoss"      # 带损失的 softmax 层，采用近似交叉熵损失函数
    top: "loss"
}

```

2. 网络结构文件 ResNet-50-validate.prototxt 的内容描述

这个文件主要是在 ResNet-50-deploy.prototxt 的基础上增加一些代码，包括增加验证集信息、增加网络损失计算层，增加计算 top-1 和 top-5 验证准确率等。下面分别进行说明。

增加验证集信息

```

layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST # 表示用于验证的数据信息
  }
  transform_param {
    mirror: false # 表示不对图像进行水平翻转
    crop_size: 224 # 表示将图片裁剪为 224×224 大小
    mean_value: 104 # 表示 B 通道的均值
    mean_value: 117 # 表示 G 通道的均值
    mean_value: 123 # 表示 R 通道的均值
  }
  data_param {
    source: "F:/dataset/ilsvcr12/val_leveladb " # 表示验证集的存放路径
    batch_size: 50 # 验证集的迷你块大小
    backend: LEVELDB # 验证集的数据格式
  }
}
# 增加网络损失计算层
layer {
  bottom: "fc1000" # 底层为全连接层
  bottom: "label" # 底层为标签
  name: "loss"
  type: "SoftmaxWithLoss" # 带损失的 softmax 层
  top: "loss"
}
# 增加计算 top-1 和 top-5 验证准确率
layer {
  bottom: "fc1000" # 计算准确率要用到全连接层 fc1000
  bottom: "label" # 计算准确率还要用到标签层
  top: "acc/top-1"
  name: "acc/top-1" # 表示 top-1 准确率层
  type: "Accuracy" # 表示该层用来计算准确率
  include {
    phase: TEST # 表示该层用来测试，这里是在验证集上测试
  }
}
layer {
  bottom: "fc1000" # 计算准确率要用到全连接层 fc1000
  bottom: "label" # 计算准确率还要用到标签层
  top: "acc/top-5"
  name: "acc/top-5" # 表示 top-5 准确率层
  type: "Accuracy" # 表示该层用来计算准确率
  include {
    phase: TEST # 表示该层用来测试，这里是在验证集上测试
  }
}
accuracy_param {

```



```
        top_k: 5                # 表示该层用来计算 top-5 准确率
    }
}
```

3. 求解器配置文件 ResNet-50-solver.prototxt 的内容描述

这个主要用来定义残差网络求解器的配置超参数，包括学习率、迭代次数、学习率策略等。文件的具体内容见方框 7.1。

方框 7.1 在 ResNet-50-solver.prototxt 中设置的超参数情况

```
net: "ResNet-50-train.prototxt"
iter_size: 2
display: 3000
base_lr: 0.05
lr_policy: "multistep"
stepvalue: 150000
stepvalue: 300000
gamma: 0.1
max_iter: 600000
momentum: 0.9
weight_decay: 0.0001
snapshot: 6000
snapshot_prefix: "resnet"
solver_mode: GPU
```

在编写 ResNet-50-train.prototxt、ResNet-50-validate.prototxt 和 ResNet-50-solver.prototxt 之后，即可对 ResNet 大规模图像案例程序进行训练和验证。训练命令如图 7.3 所示，训练的中间结果如图 7.4 所示，训练的最终结果如图 7.5 所示。验证命令如图 7.6 所示，验证结果如图 7.7 所示。

从图 7.4 可以看出，在训练到 291 000 次时，学习率为 0.005，训练损失为 2.615 72。从图 7.5 可以看出，在训练到 585 000 次时的学习率为 0.0005，在训练到 600 000 次结束时的损失为 1.5887。

从图 7.7 可以看出，第 999 个迷你块的 top-1 验证准确率为 0.66，top-5 验证准确率为 0.9，验证损失为 1.286 69。最后几行表明，总体上的 top-1 验证准确率为 0.598 92 ($\approx 59.89\%$)，top-5 验证准确率为 0.831 502 ($\approx 83.15\%$)，验证损失为 1.776。

注意 1：在图 7.3 中，“-gpu=0, 2”表示使用编号为 0 和 2 的两个 GPU 卡。

注意 2：在图 7.6 中，给验证命令设置的迭代次数为 iterations=1000，是因为在验证网络结构文件中把迷你块的大小设置为 batch_size=50。这样才能保证刚好验证 50 000 幅图像。读者也可选择不同的迷你块大小和迭代次数进行验证，但应保证 iterations \times batch_size= 验证图像总数，否则可能出错。

注意 3：卷积层的初始化方式、网络训练及验证参数的设置，只是作者的经验，并不一

定是最佳的,有兴趣的读者可进一步选择不同的参数对网络进行训练和验证。

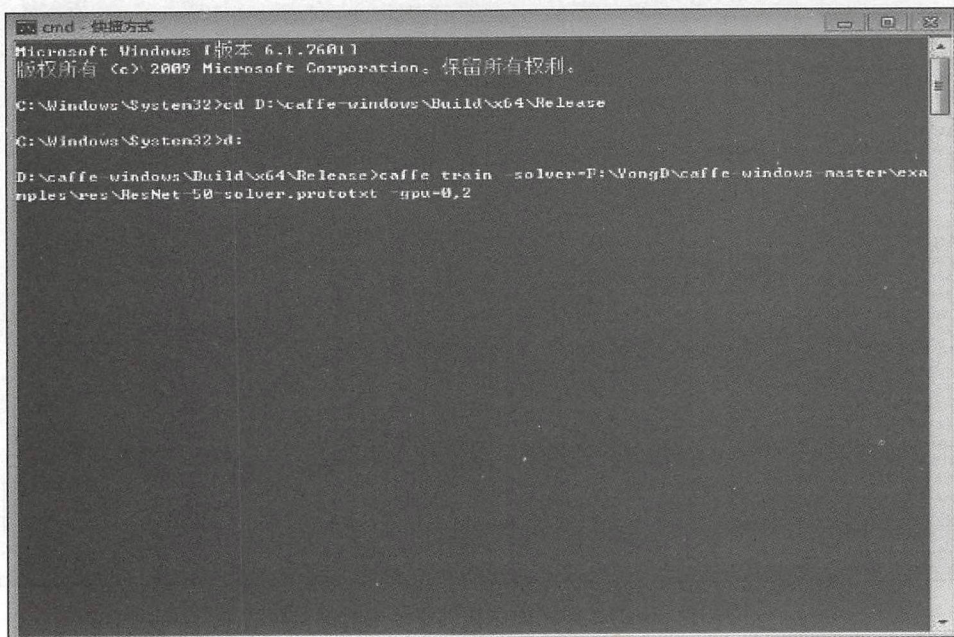


图 7.3 ResNet 大规模图像分类案例程序的训练命令

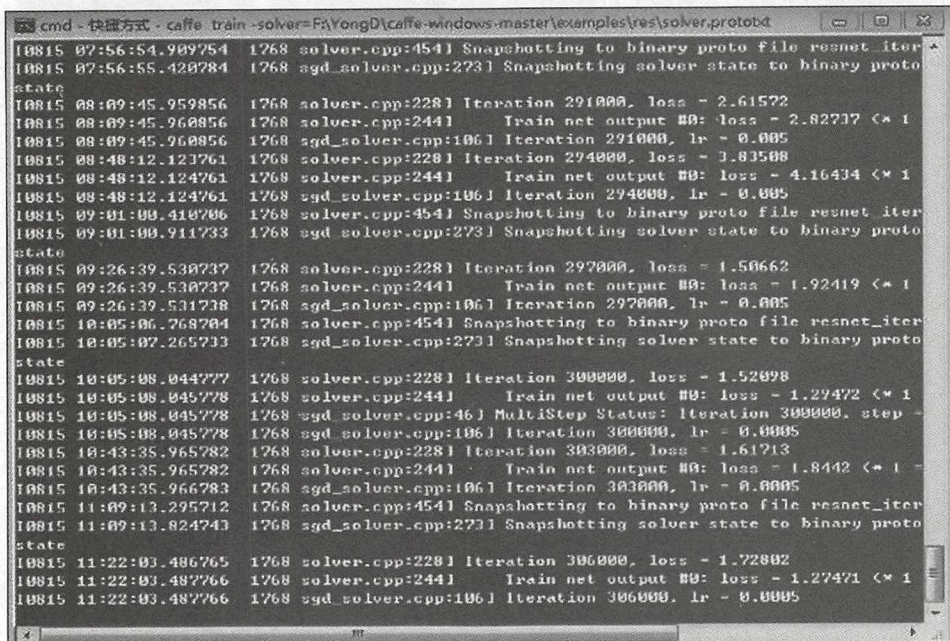
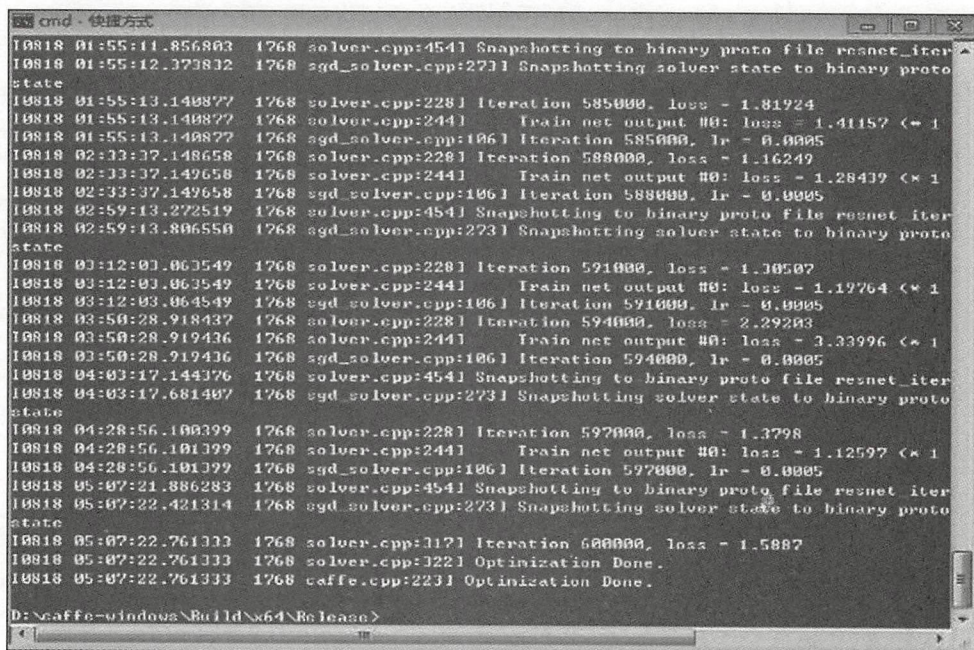


图 7.4 ResNet 案例程序在训练 291 000 ~ 306 000 次时的中间结果



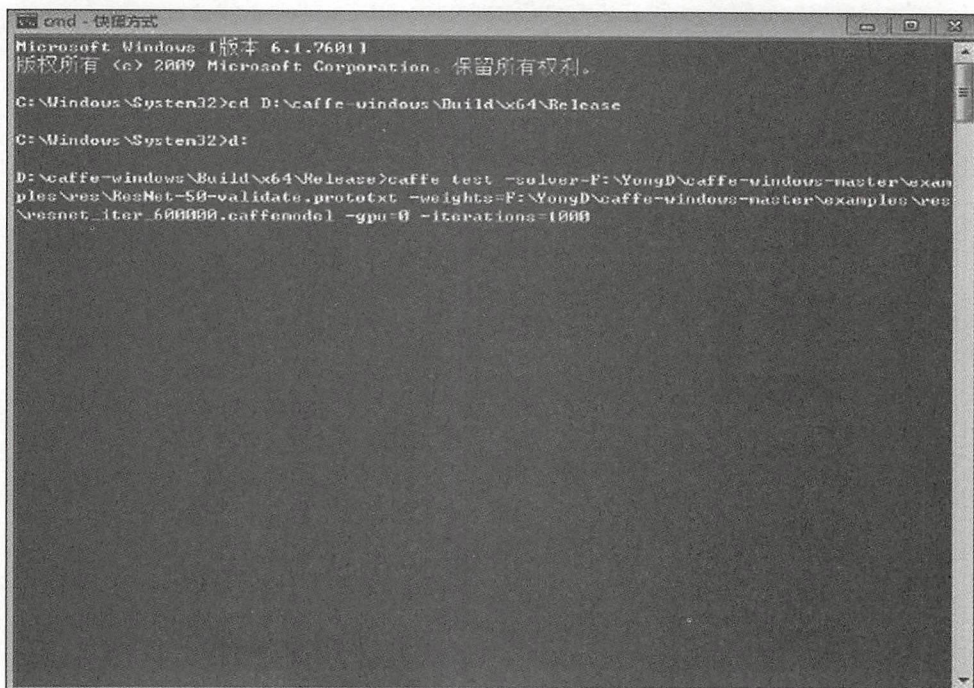
```

cmd - 快捷方式
10818 01:55:11.856803 1768 solver.cpp:454] Snapshotting to binary proto file resnet_iter
10818 01:55:12.373832 1768 sgd_solver.cpp:273] Snapshotting solver state to binary proto
state
10818 01:55:13.140877 1768 solver.cpp:228] Iteration 585000, loss = 1.81924
10818 01:55:13.140877 1768 solver.cpp:244] Train net output #0: loss = 1.41157 (* 1
10818 01:55:13.140877 1768 sgd_solver.cpp:106] Iteration 585000, lr = 0.0005
10818 02:13:37.148658 1768 solver.cpp:228] Iteration 588000, loss = 1.16249
10818 02:13:37.149658 1768 solver.cpp:244] Train net output #0: loss = 1.28437 (* 1
10818 02:13:37.149658 1768 sgd_solver.cpp:106] Iteration 588000, lr = 0.0005
10818 02:59:13.272519 1768 solver.cpp:454] Snapshotting to binary proto file resnet_iter
10818 02:59:13.806550 1768 sgd_solver.cpp:273] Snapshotting solver state to binary proto
state
10818 03:12:03.063549 1768 solver.cpp:228] Iteration 591000, loss = 1.10507
10818 03:12:03.063549 1768 solver.cpp:244] Train net output #0: loss = 1.12764 (* 1
10818 03:12:03.064549 1768 sgd_solver.cpp:106] Iteration 591000, lr = 0.0005
10818 03:50:28.918437 1768 solver.cpp:228] Iteration 594000, loss = 2.29203
10818 03:50:28.919436 1768 solver.cpp:244] Train net output #0: loss = 3.33996 (* 1
10818 03:50:28.919436 1768 sgd_solver.cpp:106] Iteration 594000, lr = 0.0005
10818 04:03:17.144376 1768 solver.cpp:454] Snapshotting to binary proto file resnet_iter
10818 04:03:17.681407 1768 sgd_solver.cpp:273] Snapshotting solver state to binary proto
state
10818 04:28:56.100399 1768 solver.cpp:228] Iteration 597000, loss = 1.3798
10818 04:28:56.101399 1768 solver.cpp:244] Train net output #0: loss = 1.12597 (* 1
10818 04:28:56.101399 1768 sgd_solver.cpp:106] Iteration 597000, lr = 0.0005
10818 05:07:21.886283 1768 solver.cpp:454] Snapshotting to binary proto file resnet_iter
10818 05:07:22.421314 1768 sgd_solver.cpp:273] Snapshotting solver state to binary proto
state
10818 05:07:22.761333 1768 solver.cpp:117] Iteration 600000, loss = 1.5887
10818 05:07:22.761333 1768 solver.cpp:322] Optimization Done.
10818 05:07:22.761333 1768 caffe.cpp:223] Optimization Done.

D:\caffe-windows\Build\x64\Release>

```

图 7.5 ResNet 案例程序在训练结束时的最终结果



```

cmd - 快捷方式
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation. 保留所有权利。

C:\Windows\System32>cd D:\caffe-windows\Build\x64\Release

C:\Windows\System32>d:

D:\caffe-windows\Build\x64\Release>caffe test -solver=F:\YongD\caffe-windows-master\exam
ples\res\ResNet-50-validate.prototxt -weights=F:\YongD\caffe-windows-master\exam
ples\resnet_iter_600000.caffemodel -gpu=0 -iterations=1000

```

图 7.6 ResNet 案例程序的验证命令


```

10818 12:38:26.054265 13564 caffe.cpp:2761 Batch 993, loss = 2.09625
10818 12:38:37.322909 13564 caffe.cpp:2761 Batch 994, acc/top-1 = 0.56
10818 12:38:37.323909 13564 caffe.cpp:2761 Batch 994, acc/top-5 = 0.8
10818 12:38:37.323909 13564 caffe.cpp:2761 Batch 994, loss = 1.74139
10818 12:38:48.535552 13564 caffe.cpp:2761 Batch 995, acc/top-1 = 0.66
10818 12:38:48.535552 13564 caffe.cpp:2761 Batch 995, acc/top-5 = 0.9
10818 12:38:48.535552 13564 caffe.cpp:2761 Batch 995, loss = 1.55472
10818 12:39:00.016207 13564 caffe.cpp:2761 Batch 996, acc/top-1 = 0.54
10818 12:39:00.016207 13564 caffe.cpp:2761 Batch 996, acc/top-5 = 0.78
10818 12:39:00.016207 13564 caffe.cpp:2761 Batch 996, loss = 2.04729
10818 12:39:11.401859 13564 caffe.cpp:2761 Batch 997, acc/top-1 = 0.66
10818 12:39:11.402858 13564 caffe.cpp:2761 Batch 997, acc/top-5 = 0.78
10818 12:39:11.402858 13564 caffe.cpp:2761 Batch 997, loss = 2.06904
10818 12:39:22.748507 13564 caffe.cpp:2761 Batch 998, acc/top-1 = 0.58
10818 12:39:22.748507 13564 caffe.cpp:2761 Batch 998, acc/top-5 = 0.86
10818 12:39:22.748507 13564 caffe.cpp:2761 Batch 998, loss = 1.62358
10818 12:39:33.907146 13564 caffe.cpp:2761 Batch 999, acc/top-1 = 0.66
10818 12:39:33.907146 13564 caffe.cpp:2761 Batch 999, acc/top-5 = 0.9
10818 12:39:33.907146 13564 caffe.cpp:2761 Batch 999, loss = 1.28669
10818 12:39:33.907146 13564 caffe.cpp:2811 Loss: 1.776
10818 12:39:33.907146 13564 caffe.cpp:2931 acc/top-1 = 0.59892
10818 12:39:33.907146 13564 caffe.cpp:2931 acc/top-5 = 0.831502
10818 12:39:33.907146 13564 caffe.cpp:2931 loss = 1.776 (* 1 = 1.776 loss)

D:\caffe-windows\Build\x64\Release>

```

图 7.7 ResNet 案例程序的验证结果

7.3 密连网络 DenseNet

7.3.1 DenseNet 的模型结构

残差网络的研究表明,如果在层间加入跨层连接,那么即使是成百上千层的网络,也能够得到准确、有效的训练。不过,残差网络一般只采用跨越 2~3 个层的跨层连接形成残差模块。密连卷积网络(Densely Connected Convolutional Network, DenseNet)^[132],简称密连网络,通过引入密连模块代替残差模块进一步扩展了残差网络的结构。与残差模块的区别在于,密连模块内部允许任意两个非相邻层之间进行跨层连接,如图 7.8 所示。因此,在极端情况,一个 L 层的密连网络,总共具有 $L(L+1)/2$ 个的层间连接,其中每层都输入之前所有层的特征图,同时把它自己的特征图都输入到之后的所有层。而一个 L 层的传统卷积网络只有 L 个层间连接,其中每层只能连向它之后的一个相邻层。密连网络有一些引人入胜的优点,比如缓和梯度消失、加强特征传播、促进特征重用和减少参数数量等。广义地说,密连网络是指包含一个或多个密连模块的卷积神经网络,如图 7.9 所示。

在密连模块中,第 l 层的特征图 x_l 是利用前面 $l-1$ 个层的特征图 x_0, x_1, \dots, x_{l-1} 来计算的,可以表达为

$$x_l = H_l([x_0, x_1, \dots, x_{l-1}]) \quad (7.9)$$

其中, $[x_0, x_1, \dots, x_{l-1}]$ 表示对第 0 层到第 $l-1$ 层产生的特征图进行张量拼接, 以方便实现。 $H_l(\cdot)$ 是一个由 3 种相继操作组成的复合函数: 块归一化 (Batch Normalization, BN), 校正线性单元 (Rectified Linear Unit, ReLU) 和 3×3 卷积 (Convolution, Conv)。注意, 在大小发生变化时, 对底层特征图进行拼接可能会出现问題。这时需要在密连网络中插入过渡层 (transition layer) 做卷积和池化, 以改变特征图的大小。通常, 过渡层由一个块归一化层、一个 1×1 卷积层和一个 2×2 平均池化层组成。

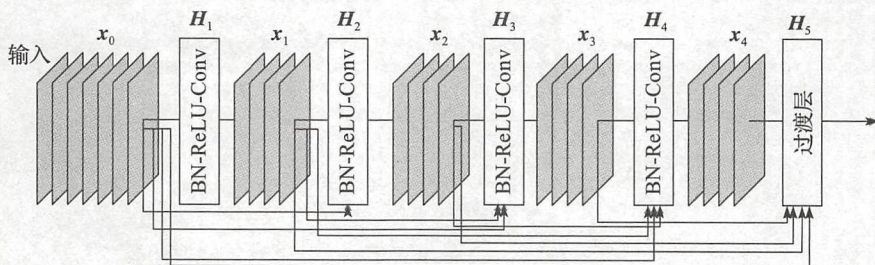


图 7.8 密连模块的结构示意图

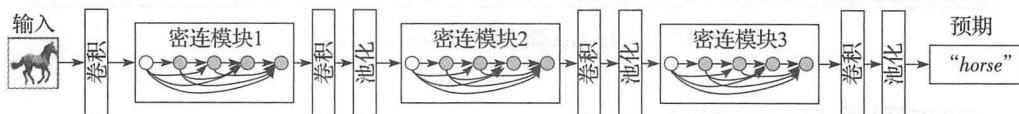


图 7.9 一个包含 3 个密连模块的密连网络

在密连模块中, 如果每个函数 H_l 都输出 k 个特征图, 那么第 l 层就有 $k(l-1) + k_0$ 个输入特征图, 其中 k_0 是输入通道的个数, k 称为生长率。为了控制网络的宽度和提高参数的效率, 超参数 k 一般被限制为小整数, 比如 $k = 12$ 。这种对生长率的控制, 既可以减少密连网络的参数, 又能够保证密连网络的性能。另外, 每一层虽然只输出 k 个特征图, 但会有更多的输入特征图。减少输入特征图的办法是在每个 3×3 卷积操作之前引入一个 1×1 卷积的瓶颈层 (bottleneck layer), 以提高计算效率。最后, 为进一步简化模型, 还可以减少过渡层的特征图数量。如果密连模块包含 m 个特征图, 可让随后的过渡层输出 $\lfloor \theta m \rfloor$ 个特征图, 其中 θ ($0 < \theta \leq 1$) 称为压缩因子 (compression factor)。 $\theta = 1$ 表示通过过渡层的特征图数量不变。

表 7.2 给出了一种用于 CIFAR-10 图像分类的密连网络结构, 总共包含 39 个卷积层、3 个池化层和 3 个密连模块。其中, 第 1 层是卷积层, 第 2 ~ 13 层构成密连模块, 第 14 层 (卷积层) 和池化层 Pooling1 构成过渡层, 第 15 ~ 26 层构成密连模块, 第 27 层 (卷积层) 和池化层 Pooling2 构成过渡层, 第 28 ~ 39 层构成密连模块, 最后是池化层 Pooling3 和软最大输出层。注意, 在这个密连网络中, 除了第 1 个卷积层 Convolution1, 其余卷积层都按概率 20% 使用了丢失输出 (dropout) 处理。

表 7.2 一种用于 CIFAR-10 图像分类的密连网络结构

层名	输出大小	卷积核大小 / 数量 / 填充列数 / 步长
Convolution1	32×32	$3 \times 3/16/1/1$
Convolution2 ~ Convolution13 (密连模块 1)	32×32	$3 \times 3/12/1/1$
Convolution14	32×32	$1 \times 1/160/0/1$
Pooling1	16×16	$2 \times 2/0/0/2$
Convolution15 ~ Convolution26 (密连模块 2)	16×16	$3 \times 3/12/1/1$
Convolution27	16×16	$1 \times 1/304/0/1$
Pooling2	8×8	$2 \times 2/0/0/1$
Convolution28 ~ Convolution39 (密连模块 3)	8×8	$3 \times 3/12/1/1$
Pooling3	1×1	$8 \times 8/0/0/0$
Softmax	1×10	

7.3.2 DenseNet 的 Caffe 代码实现及说明

一般情况下,密连网络的结构相对复杂,难以实现通用代码。但对于表 7.2 描述的密连网络,已经有公开的 Caffe 实现代码,下载地址为 <https://github.com/liuzhuang13/DenseNetCaffe>。在此地址的文件夹下,包含 3 个重要文件:train_densenet.prototxt、test_densenet.prototxt 和 solver.prototxt。train_densenet.prototxt 定义了用于训练的网络结构,test_densenet.prototxt 定义了用于测试的网络结构,solver.prototxt 定义了求解器的配置超参数。

下面参考表 7.2,对 train_densenet.prototxt 的关键实现代码进行说明,包括卷积层 Convolution1 和 Convolution2、拼接层 Concat1 和全局池化层 Pooling3。

1. 卷积层 Convolution1 的实现代码及说明

```

layer {
  name: "Convolution1"
  type: "Convolution"      # 表示该层为卷积层
  bottom: "Data1"          # 底层为输入数据
  top: "Convolution1"
  convolution_param {
    num_output: 16          # 输出特征图的个数
    bias_term: false        # 表示不学习偏置
    pad: 1                  # 对输入的四边各扩展 1 个单位长度,并用 0 填充
    kernel_size: 3          # 卷积核的大小
    stride: 1               # 卷积核的步长
    weight_filler {
      type: "msra"
    }
    bias_filler {
      type: "constant"
    }
  }
}

```



```

layer {
  name: "BatchNorm1"
  type: "BatchNorm"           # 表示该层是块归一化层
  bottom: "Convolution1"
  top: "BatchNorm1"
  param {
    lr_mult: 0
    decay_mult: 0
  }
  param {
    lr_mult: 0
    decay_mult: 0
  }
  param {
    lr_mult: 0
    decay_mult: 0
  }
}
layer {
  name: "Scale1"
  type: "Scale"               # 表示该层为尺度变换层
  bottom: "BatchNorm1"
  top: "BatchNorm1"
  scale_param {
    filler {
      value: 1                 # 表示将变换因子初始化为 1
    }
    bias_term: true           # 表示在学习变换因子的同时，还学习偏置
    bias_filler {
      value: 0                 # 表示将偏置初始化为 0
    }
  }
}
layer {
  name: "ReLU1"
  type: "ReLU"                # 把激活函数选为 ReLU
  bottom: "BatchNorm1"
  top: "BatchNorm1"
}

```

2. 卷积层 Convolution2 的实现代码及说明

```

layer {
  name: "Convolution2"
  type: "Convolution"
  bottom: "BatchNorm1"
  top: "Convolution2"
  convolution_param {
    num_output: 12
    bias_term: false
    pad: 1
  }
}

```

```

        kernel_size: 3
        stride: 1
        weight_filler {
            type: "msra"
        }
        bias_filler {
            type: "constant"
        }
    }
}
layer {
    name: "Dropout1"
    type: "Dropout"           # 表示该层为丢失输出层
    bottom: "Convolution2"
    top: "Dropout1"
    dropout_param {
        dropout_ratio: 0.2    # 表示丢失概率为 20%
    }
}
}

```

3. 拼接层 Concat1 的实现代码及说明

```

layer {
    name: "Concat1"
    type: "Concat"           # 表示该层为拼接层
    bottom: "Convolution1"   # 该层的第 1 个输入为 Convolution1 层
    bottom: "Dropout1"       # 该层的第 2 个输入为 Dropout1 层
    top: "Concat1"
    concat_param {
        axis: 1
    }
}
layer {
    name: "BatchNorm2"
    type: "BatchNorm"        # 表示该层是块归一化层
    bottom: "Concat1"
    top: "BatchNorm2"
    param {
        lr_mult: 0
        decay_mult: 0
    }
    param {
        lr_mult: 0
        decay_mult: 0
    }
    param {
        lr_mult: 0
        decay_mult: 0
    }
}
layer {
    name: "Scale2"

```

```

type: "Scale"                # 表示该层是尺度变换层
bottom: "BatchNorm2"
top: "BatchNorm2"
scale_param {
  filler {
    value: 1
  }
  bias_term: true
  bias_filler {
    value: 0
  }
}
}
layer {
  name: "ReLU2"
  type: "ReLU"                # 表示将激活函数选为 ReLU
  bottom: "BatchNorm2"
  top: "BatchNorm2"
}

```

4. 全局池化 Pooling3 的实现代码及说明

```

layer {
  name: "Pooling3"
  type: "Pooling"
  bottom: "BatchNorm39"
  top: "Pooling3"
  pooling_param {
    pool: AVE                  # 表示池化方式为平均池化
    global_pooling: true      # 表示使用全局池化
  }
}

```

7.3.3 DenseNet 的物体图像分类案例

本节描述一个利用 DenseNet 进行物体图像分类的案例，其中用到的 CIFAR-10 数据集可以根据表 1.2 提供的地址下载。注意，这个数据集的图像是 .jpg 格式的，需要参考本书 3.6.1 节转换成 Caffe 支持的 LEVELDB 格式。此外，这个案例还需要在 train_densenet.prototxt 和 test_densenet.prototxt 文件中分别修改训练集和测试集的路径信息，并在求解器配置文件 solver.prototxt 中把网络结构文件路径修改为 train_densenet.prototxt 的路径。下面分别进行描述。

1. 修改 train_densenet.prototxt 的均值文件路径和训练集存放路径

将该文件中的均值文件路径和训练集存放路径，修改为用户计算机的相应路径，例如：

```

layer {
  name: "Data1"

```



```
type: "Data"
top: "Data1"
top: "Data2"
transform_param {
  mean_file: "F:/DenseNetCaffe-master/mean.binaryproto" # 均值文件路径
}
data_param {
  source: "F:/DenseNetCaffe-master/cifar10_train_leveldb" # 训练集存放路径
  batch_size: 64
  backend: LEVELDB
}
}
```

注意：均值文件是使用训练集计算得到的，测试和训练阶段使用相同的均值文件。对于训练图像，使用转换成 LEVELDB 格式后的 cifar10_train_leveldb 文件，按照图 7.10 所示的命令，即可得到均值文件。

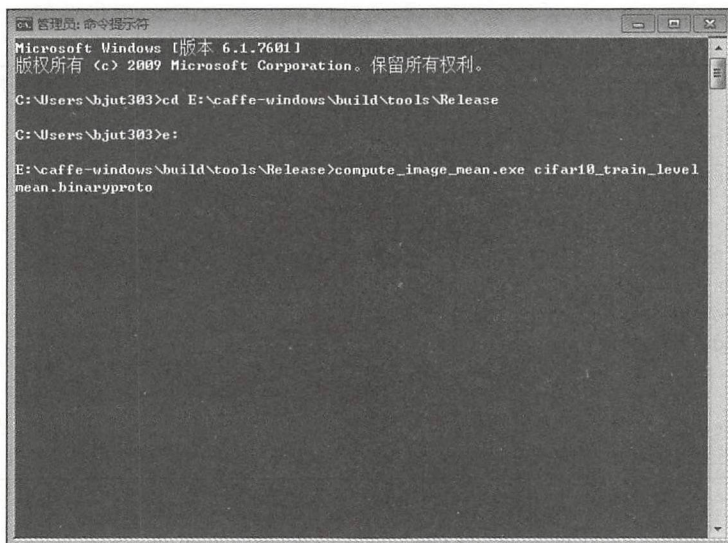


图 7.10 计算均值文件的命令

2. 修改 test_densenet.prototxt 的均值文件路径和测试集存放路径

将该文件中的均值文件路径和测试集存放路径，修改为用户计算机的相应路径，例如：

```
layer {
  name: "Data1"
  type: "Data"
  top: "Data1"
  top: "Data2"
```

```

transform_param {
  mean_file: "F:/DenseNetCaffe-master/mean.binaryproto" # 均值文件路径
}
data_param {
  source: "F:/DenseNetCaffe-master/cifar10_test_leveldb" # 测试集存放路径
  batch_size: 50
  backend: LEVELDB
}
}

```

注意：test_densenet.prototxt 与 train_densenet.prototxt 具有相同的均值文件路径，其中的均值信息都是利用训练集计算的。

3. 修改 solver.prototxt 的配置超参数

这个文件的大部分配置超参数可以直接使用，但读者可能需要修改网络结构文件的存放路径和训练结果参数文件的存放路径，具体内容见方框 7.2。

方框 7.2 在 solver.prototxt 中设置的超参数情况

```

net: "F:/DenseNetCaffe-master/train_densenet.prototxt" # 网络结构文件的存放路径
base_lr: 0.1
display: 200
max_iter: 230000
lr_policy: "multistep"
gamma: 0.1
momentum: 0.9
weight_decay: 0.0001
solver_mode: GPU
random_seed: 831486
stepvalue: 115000
stepvalue: 172500
type: "Nesterov"
snapshot_prefix: "F:/DenseNetCaffe-master/dense" # 训练结果参数文件的存放路径

```

4. DenseNet 案例程序的运行演示过程

在修改 train_densenet.prototxt、test_densenet.prototxt 和 solver.prototxt 之后，即可对 DenseNet 进行训练和测试。训练命令如图 7.11 所示，训练的中间结果如图 7.12 所示，最终结果如图 7.13 所示。测试命令如图 7.14 所示，测试结果如图 7.15 所示。

从图 7.12 可以看出，在训练到 140 000 次时，学习率为 0.01，训练准确率为 100%，训练损失为 0.025 742 3。从图 7.13 可以看出，在训练到 229 800 次时的学习率为 0.001，在训练到 230 000 次结束时的损失为 0.001 697 34。

从图 7.15 可以看出，第 199 个迷你块的测试准确率为 0.92，测试损失为 0.341 569。最后几行表明，总体上的测试准确率为 0.9266 (92.66%)，测试损失为 0.300 462。

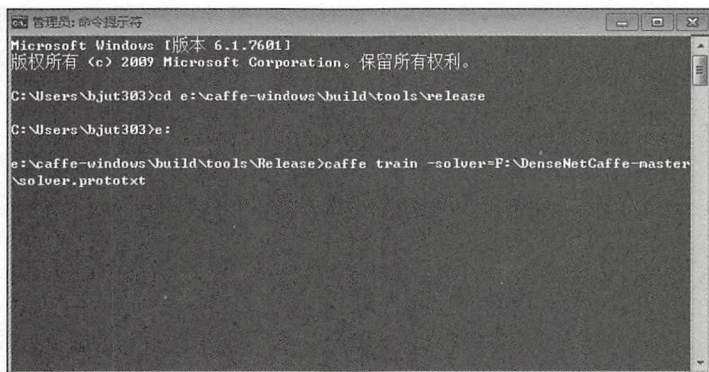


图 7.11 DenseNet 物体图像分类案例程序的训练运行命令

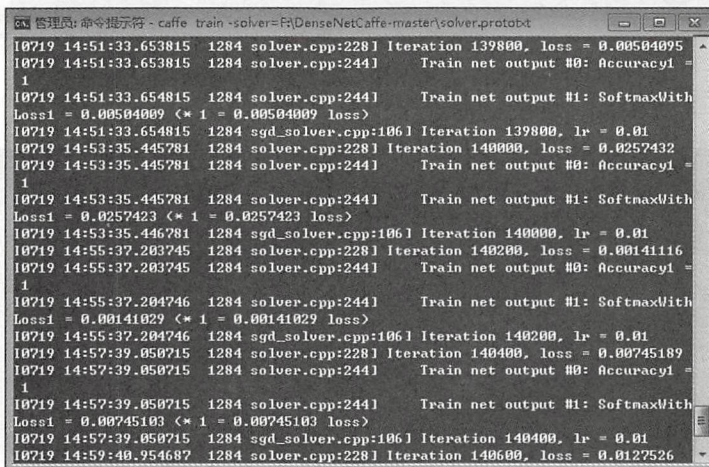


图 7.12 DenseNet 物体图像分类案例程序训练 139 800 ~ 140 400 次时的中间结果

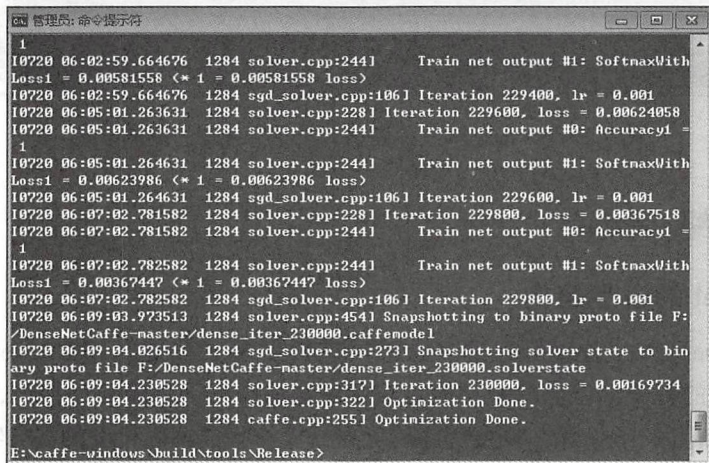


图 7.13 DenseNet 物体图像分类案例程序在训练结束时的最终结果


```

管理员: 命令提示符
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\bjut303>cd e:\caffe-windows\build\tools\Release

C:\Users\bjut303>e:

e:\caffe-windows\build\tools\Release>caffe test -model=F:\DenseNetCaffe-master\
est_densenet.prototxt -weights=F:\DenseNetCaffe-master\dense_iter_230000.caffemo
del -iterations=200

```

图 7.14 DenseNet 物体图像分类案例程序的测试运行命令

```

管理员: 命令提示符
7
10720 10:57:21.911900 3732 caffe.cpp:3091 Batch 194, Accuracy1 = 0.94
10720 10:57:21.911900 3732 caffe.cpp:3091 Batch 194, SoftmaxWithLoss1 = 0.24164
7
10720 10:57:28.947301 3732 caffe.cpp:3091 Batch 195, Accuracy1 = 0.9
10720 10:57:28.948302 3732 caffe.cpp:3091 Batch 195, SoftmaxWithLoss1 = 0.42381
4
10720 10:57:35.929702 3732 caffe.cpp:3091 Batch 196, Accuracy1 = 0.94
10720 10:57:35.929702 3732 caffe.cpp:3091 Batch 196, SoftmaxWithLoss1 = 0.23047
5
10720 10:57:42.962103 3732 caffe.cpp:3091 Batch 197, Accuracy1 = 0.94
10720 10:57:42.962103 3732 caffe.cpp:3091 Batch 197, SoftmaxWithLoss1 = 0.18121
7
10720 10:57:49.948503 3732 caffe.cpp:3091 Batch 198, Accuracy1 = 0.98
10720 10:57:49.948503 3732 caffe.cpp:3091 Batch 198, SoftmaxWithLoss1 = 0.03973
53
10720 10:57:56.953903 3732 caffe.cpp:3091 Batch 199, Accuracy1 = 0.92
10720 10:57:56.954903 3732 caffe.cpp:3091 Batch 199, SoftmaxWithLoss1 = 0.34156
9
10720 10:57:56.954903 3732 caffe.cpp:3141 Loss: 0.300462
10720 10:57:56.954903 3732 caffe.cpp:3261 Accuracy1 = 0.9266
10720 10:57:56.954903 3732 caffe.cpp:3261 SoftmaxWithLoss1 = 0.300462 (* 1 = 0.
300462 loss)

e:\caffe-windows\build\tools\Release>

```

图 7.15 DenseNet 物体图像分类案例程序的测试结果

7.4 拼接网络 CatNet

7.4.1 CatNet 的模型结构

密连网络 DenseNet 通过引入更多的跨层连接发展了残差网络 ResNet 的结构，虽然获得了综合性能的提升，但仍然有必要对某些特殊的跨连网络做专门探讨，比如拼接网络 CatNet^[4]，结构如图 7.16 所示。拼接网络的优点是能够集成不同尺度的图像特征进行分类和识别。

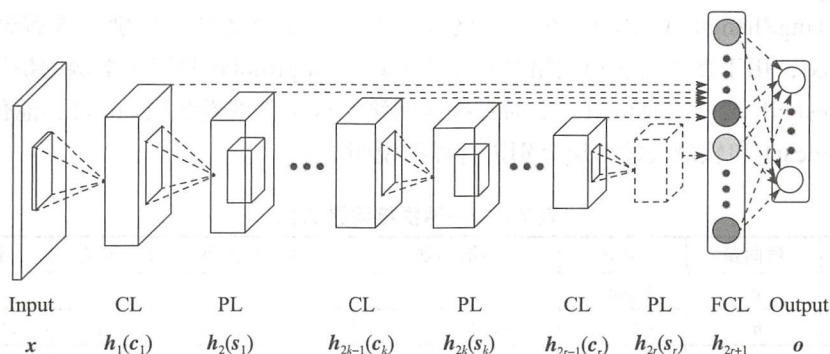


图 7.16 拼接网络的结构意图

从图 7.16 不难看出, 拼接网络包含 r 个交错的卷积层和池化层, 再跟一个全连接层和输出层。其中, 全连接层是所有卷积层和池化层通过跨层连接拼接得到的。在拼接网络中, 输入 x 是一个三维数组, 大小为 $h \times w \times n$, 其中 h 和 w 是空间维度, n 是通道维度, $n = 3$ 表示彩色图像, $n = 1$ 表示灰度图像。卷积层的计算可表示为

$$h_{2k-1,j} = c_{k,j} = f(u_{2k-1,j}) = f\left(\sum_i h_{2k-2,i} * W_{ij}^{2k-1} + b_j^{2k-1}\right), 1 \leq k \leq r \quad (7.10)$$

其中, W_{ij}^{2k-1} 表示第 $(2k-2)$ 个隐含层的第 i 个特征面 $h_{2k-2,i}$ 与第 $(2k-1)$ 个隐含层的第 j 个特征面 $h_{2k-1,j}$ 之间的卷积核, b_j^{2k-1} 表示相应的偏置, $c_{k,j}$ 表示第 k 个卷积层的第 j 个特征面。注意, $h_0 = x$ 。

在拼接网络中, 所有池化层都采用不重叠的 2×2 窗口池化, 表示为

$$h_{2k,j} = s_{k,j} = \text{pooling}\{h_{2k-1,j}\}, 1 \leq k \leq r \quad (7.11)$$

其中, $\text{pooling}\{\cdot\}$ 可以是平均池化 (average pooling) 或者最大池化 (max-pooling)。 $s_{k,j}$ 表示第 k 个池化层的第 j 个特征面。

全连接层是若干个卷积层和池化层的激活通过跨层连接产生的拼接向量, 具有如下形式:

$$h_{2r+1} = (a_1 h_1, a_2 h_2, \dots, a_{2k-1} h_{2k-1}, a_{2k} h_{2k}, \dots, h_{2r}) \quad (7.12)$$

跨连指示符定义为 $SI = a_1 a_2 a_3 \dots a_{2r-1} a_{2r}$ 。 SI 是一个 0 和 1 构成的字符串, 表示跨层连接的方式。例如, $SI = 111 \dots 1$ 表示所有卷积层和池化层都跨连到全连接层。 $SI = 100 \dots 0$ 表示只有第一个卷积层跨连到全连接层。 $SI = 000 \dots 0$ 表示没有跨层连接, 即表示标准的 CNN。

输出层是一个 C 维的软最大 (softmax) 函数, 预测 C 个不同类别的概率分布, 表示为

$$o = \text{softmax}(u) = \text{softmax}(W^{2r+2} h_{2r+1} + b^{2r+2}) \quad (7.13)$$

其中, W^{2r+2} 和 b^{2r+2} 分别是权值和偏置。

7.4.2 CatNet 的 Caffe 代码实现及说明

拼接网络是一种框架结构, 可以具体实现为多种不同的跨连形式。这里仅针对表 7.3 的特定拼接网络 (跨连指示符为 $SI = 01110$), 给出 Caffe 实现代码, 下载地址为 <https://>

github.com/TingZhang08/CatNet。在这个网址下共包含 3 个文件，分别为求解器配置文件 solver.prototxt，用于参考的普通网络结构文件 train_test.prototxt 和用于案例的拼接网络结构文件 train_test_s1_c2_s2.prototxt。下面对拼接网络结构文件的卷积层 conv1、池化层 pool1、全连接层 concat1 和软最大输出层分别进行详细说明。

表 7.3 一种拼接网络结构

层名	层向量	类型	激活函数	卷积核大小	步长	输出大小
data	x	Input				$32 \times 32 \times 1$
conv1	h_1	CL	ReLU	5×5	1	$28 \times 28 \times 6$
pool1	h_2	PL	max-pooling	2×2	2	$14 \times 14 \times 6$
conv2	h_3	CL	ReLU	5×5	1	$10 \times 10 \times 12$
pool2	h_4	PL	max-pooling	2×2	2	$5 \times 5 \times 12$
conv3	h_5	CL	ReLU	2×2	1	$4 \times 4 \times 16$
pool3	h_6	PL	max-pooling	2×2	2	$2 \times 2 \times 16$
concat1	h_7	FCL				2 740
softmax	o	Output	softmax			2

1. 卷积层 conv1 的实现代码及说明

```

layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 6
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"

```

表示该层为卷积层
底层为输入数据
表示卷积核的学习率系数
表示偏置的学习率系数
输出特征图的个数
卷积核的大小
卷积核的步长
表示权值的初始化方法
表示偏置的初始化


```
    top: "conv1"  
}
```

2. 池化层 pool1 的实现代码及说明

```
layer {  
  name: "pool1"  
  type: "Pooling"  
  bottom: "conv1"  
  top: "pool1"  
  pooling_param {  
    pool: MAX      # 把池化方式选为最大池化  
    kernel_size: 2  # 池化窗口的大小  
    stride: 2       # 池化窗口的步长  
  }  
}
```

3. 全连接层 concat1 的实现代码及说明

```
layer {  
  name: "pool1_flatt"  
  type: "Flatten"      # 表示将第一个池化层 pool1 展开成一个向量  
  bottom: "pool1"  
  top: "pool1_flatt"  
}  
  
layer {  
  name: "conv2_flatt"  
  type: "Flatten"      # 表示将第二个卷积层 conv2 展开成一个向量  
  bottom: "conv2"  
  top: "conv2_flatt"  
}  
  
layer {  
  name: "pool2_flatt"  
  type: "Flatten"      # 表示将第二个池化层 pool2 展开成一个向量  
  bottom: "pool2"  
  top: "pool2_flatt"  
}  
  
layer {  
  name: "pool3_flatt"  
  type: "Flatten"      # 表示将第三个池化层 pool3 展开成一个向量  
  bottom: "pool3"  
  top: "pool3_flatt"  
}  
  
layer {  
  name: "concat1"  
  bottom: "pool1_flatt"  
  bottom: "conv2_flatt"
```

```

bottom: "pool2_flatt"
bottom: "pool3_flatt"
top: "concat1"
type: "Concat"          # 表示将 pool1、conv2、pool2 和 pool3 进行拼接
concat_param {
    axis: 1              # 表示按照通道进行拼接
}

```

4. 软最大输出层的实现代码及说明

```

layer {
    name: "ip2"
    type: "InnerProduct" # 表示该层为全连接层
    bottom: "ip1"
    top: "ip2"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    inner_product_param {
        num_output: 2
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
}
layer {
    name: "accuracy"
    type: "Accuracy"
    bottom: "ip2"
    bottom: "label"
    top: "accuracy"
    include {
        phase: TEST          # 表示计算测试集的分类准确率
    }
}
}
layer {
    name: "accuracy1"
    type: "Accuracy"
    bottom: "ip2"
    bottom: "label"
    top: "accuracy"
    include {
        phase: TRAIN        # 表示计算训练集的分类准确率
    }
}
}
layer {

```

```

    name: "loss"
    type: "SoftmaxWithLoss"      # 表示计算最大损失
    bottom: "ip2"
    bottom: "label"
    top: "loss"
}

```

7.4.3 CatNet 的人脸图像性别分类案例

本节描述一个利用 CatNet 进行性别分类的案例，其中用到的 AR 数据集可以根据表 1.2 提供的地址下载。注意，这个数据集的图像是 .jpg 格式的，需要参考本书 3.6.1 节转换成 Caffe 支持的 LEVELDB 格式。此外，这个案例不仅需要编写 train_test_s1_c2_s2.prototxt 文件，还需要编写配置文件 solver.prototxt。下面分别进行描述。

1. 编写 train_test_s1_c2_s2.prototxt 文件

```

name: "CatNet"
layer {
  name: "ar"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "D:/YongD/caffe-windows-master/examples/ar/train_leveldb"
    backend: LEVELDB
    batch_size: 100
  }
  transform_param {
    scale: 0.00390625
  }
  include: { phase: TRAIN }
}
layer {
  name: "ar"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "D:/YongD/caffe-windows-master/examples/ar/test_leveldb"
    backend: LEVELDB
    batch_size: 100
  }
  transform_param {
    scale: 0.00390625
  }
  include: { phase: TEST }
}

layer {
  name: "conv1"
  type: "Convolution"

```



```
bottom: "data"
top: "conv1"
param {
  lr_mult: 1
}
param {
  lr_mult: 2
}
convolution_param {
  num_output: 6
  kernel_size: 5
  stride: 1
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
  }
}
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 12
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
  }
}
```

```
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu2"
  type: "ReLU"
  bottom: "conv2"
  top: "conv2"
}
layer {
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "conv3"
  type: "Convolution"
  bottom: "pool2"
  top: "conv3"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 16
    kernel_size: 2
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu3"
  type: "ReLU"
  bottom: "conv3"
  top: "conv3"
}
layer {
  name: "pool3"
```

```
    type: "Pooling"
    bottom: "conv3"
    top: "pool3"
    pooling_param {
      pool: MAX
      kernel_size: 2
      stride: 2
    }
  }
  layer {
    name: "pool1_flatt"
    type: "Flatten"
    bottom: "pool1"
    top: "pool1_flatt"
  }
  layer {
    name: "conv2_flatt"
    type: "Flatten"
    bottom: "conv2"
    top: "conv2_flatt"
  }
  layer {
    name: "pool2_flatt"
    type: "Flatten"
    bottom: "pool2"
    top: "pool2_flatt"
  }
  layer {
    name: "pool3_flatt"
    type: "Flatten"
    bottom: "pool3"
    top: "pool3_flatt"
  }
  layer {
    name: "concat1"
    bottom: "pool1_flatt"
    bottom: "conv2_flatt"
    bottom: "pool2_flatt"
    bottom: "pool3_flatt"
    top: "concat1"
    type: "Concat"
    concat_param {
      axis: 1
    }
  }
  layer {
    name: "ip1"
    type: "InnerProduct"
    bottom: "concat1"
    top: "ip1"
    param {
      lr_mult: 1
    }
  }
  param {
```



```

        lr_mult: 2
    }
    inner_product_param {
        num_output: 2
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
layer {
    name: "accuracy"
    type: "Accuracy"
    bottom: "ip1"
    bottom: "label"
    top: "accuracy"
    include {
        phase: TEST
    }
}
layer {
    name: "accuracy1"
    type: "Accuracy"
    bottom: "ip1"
    bottom: "label"
    top: "accuracy"
    include {
        phase: TRAIN
    }
}
}
layer {
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "ip1"
    bottom: "label"
    top: "loss"
}
}

```

2. 编写 solver.prototxt 文件

该文件指定了学习率、迭代次数、学习策略等，具体内容见方框 7.3。

在编写 train_test_sl_c2_s2.prototxt 和 solver.prototxt 之后，即可对 CatNet 进行训练和测试，具体步骤为，进入 DOS 命令窗口，在命令窗口输入如图 7.17 所示的命令，即可运行程序，开始训练和测试网络。程序运行过程中，随着迭代次数的增加，网络的训练损失和准确率也发生变化。如图 7.18 所示，在迭代次数为 2400 时，学习率为 0.001，训练损失约为 0.063 56，训练准确率为 99%，测试准确率为 93.67%。如图 7.19 所示，在程序运行结束时，训练损失约为 0.028 36，训练准确率为 100%，测试准确率为 94.83%。

方框 7.3 在 solver.prototxt 中设置的超参数情况

```

net: "D:/YongD/caffe-windows-master/examples/ar/train_test_s1_c2_s2.prototxt"
test_iter: 100
test_interval: 200
base_lr: 0.001
momentum: 0.9
weight_decay: 0.004
lr_policy: "fixed"
display: 200
max_iter: 5000
snapshot: 5000
snapshot_format: HDF5
snapshot_prefix: "D:/YongD/caffe-windows-master/examples/ar/ar_full"
solver_mode: GPU

```

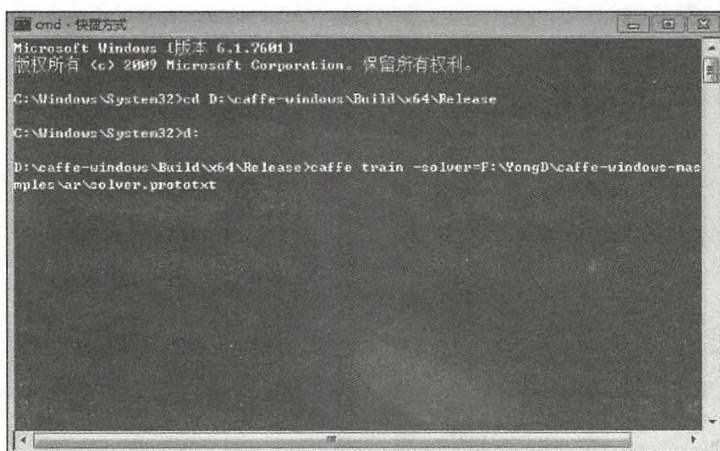


图 7.17 CatNet 人脸图像性别分类案例的训练运行命令

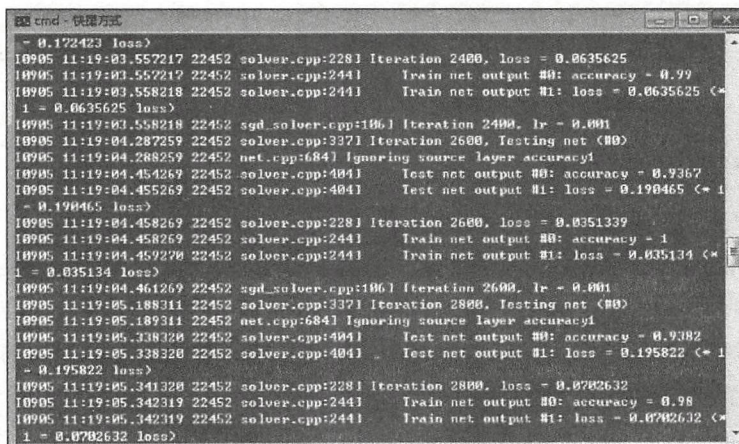


图 7.18 CatNet 人脸图像性别分类案例程序运行的中间结果

卷积神经网络的区域模型

卷积神经网络在大规模图像分类问题上的成功促使人们考虑将其应用于目标检测的问题。与图像的分类任务不同，目标检测需要从图像中检测并定位特定的多个目标（即物体或对象）。利用卷积网络进行目标检测的基本思路是先推荐候选区域，再利用卷积网络对候选区域分类。本章主要讨论这些区域模型的形成和发展，主要包括：区域卷积网络 R-CNN、快速区域卷积网络 Fast R-CNN、更快区域卷积网络 Faster R-CNN、你只看一次网络 YOLO 和单次检测器 SSD。不仅依次介绍它们的模型结构，而且分析了后三者的 TensorFlow 代码及相应的图像目标检测案例。

8.1 区域卷积网络 R-CNN

卷积神经网络在 ImageNet 大规模视觉识别挑战赛（ILSVRC）中显著提高了图像分类的准确率，所取得的巨大成功引起了广泛关注，同时也引起了各种争议。争议的中心问题是：卷积神经网络对 ImageNet 的分类结果能够在多大程度上推广应用于 PASCAL VOC 挑战赛的目标检测？为了回答这个问题，有人提出了区域卷积神经网络（R-CNN）的新模型^[134]。

R-CNN 是一种目标检测模型。与图像分类不同，目标检测要求在图像中确定多个可能目标的位置。

R-CNN 采用了滑动窗口的策略来进行定位，在“利用区域识别（recognition using region）”的思想指导下，取得了目标检测和语义分割的成功。图 8.1 是利用 R-CNN 进行目标检测的过程，其中包括 3 个关键模块：区域推荐、特征提取和区域分类。这三个模块的详细说明如下。

1) 第一个模块是区域推荐（region proposal），用来给输入图像生成约 2000 个类别无关的区域推荐构成候选检测集。R-CNN 采用的区域推荐方法是选择性搜索（selective search）^[135]，其他推荐方法包括目标构成度（objectness）^[136]、类别无关目标推荐（category-independent object proposal）^[137]、受限参数最小割（Constrained Parametric Min-cut, CPMC）^[138]等。

2) 第二个模块是**特征提取**, 用来从每个区域推荐中提取固定长度为 4096 的特征向量。R-CNN 利用卷积网络 (比如 AlexNet) 来计算每个推荐的特征, 要求先把推荐转变成为卷积网络的输入大小 (比如 227×227 的 RGB 图像), 并且做减均值处理。

3) 第三个模块是**区域分类**, 用来对每个推荐区域进行打分和筛选。R-CNN 先采用类别相关线性支持向量机 (SVM) 对所提取的特征向量打分, 再根据分值高低, 通过贪婪非最大抑制策略进行筛选, 保留高分推荐, 拒绝低分推荐, 拒绝条件是与其某高分推荐的交并比 (IoU) 大于一个通过学习得到的阈值。

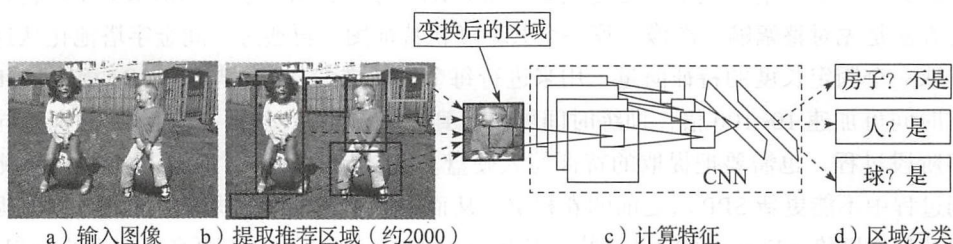


图 8.1 R-CNN 的目标检测过程: a) 输入一幅图像; b) 自底向上提取大约 2000 个推荐区域并变换为固定大小; c) 用 CNN 计算每个推荐区域的特征; d) 用类别相关线性支持向量机对推荐区域进行分类

在训练过程中, R-CNN 还会面临标注数据稀少的问题, 解决办法包括 3 个有效步骤: 有监督预训练、领域相关微调和目标类别分类。**有监督预训练**是在一个大数据集 (比如 ImageNet) 上对卷积网络进行判别预训练, 但只有图像水平的标注, 没有边框标记。**领域相关微调**是在校正区域推荐上对卷积网络继续进行微调训练, 但需要将原来的 N 路分类层修改为 $N+1$ 路分类层 (其中 N 是目标类别数, 加 1 个背景类别, 对 ImageNet 而言, $N=1000$), 微调所用的推荐正例要求与真实边框的交并比不低于 0.5, 其余的都是背景反例。**目标类别分类**就是每类训练一个线性支持向量机, 其中每类的正例简单定义为真实边框包围的图像区域, 反例定义为与该类所有真实边框的交并比都低于重叠阈值 (比如 0.3) 的图像区域, 但控制反例数量需要使用标准硬反例挖掘方法 (standard hard negative mining method)^[139]。

区域卷积网络 R-CNN 在 200 类的 ILSVRC 2013 检测数据集上, 可以达到 31.4% 的平均准确率 (mAP), 远远高于 OverFeat 在竞赛后提交的第二好结果 24.3%^[140]。R-CNN 的成功关键在于两点: 一是利用了卷积网络, 二是利用了 3 个有效训练步骤。R-CNN 是后续许多目标定位和检测方法的基础。

8.2 快速区域卷积网络 Fast R-CNN

R-CNN 利用深层卷积网络对目标区域推荐分类, 可以达到较高的检测精度, 但也有以下明显不足:

量1) 训练过程阶段多。R-CNN 首先用 log 损失函数在区域推荐上对卷积网络进行微调, 然后用支持向量机拟合卷积特征得到目标检测器, 代替微调学到的软最大分类器。最后才进行边框回归学习。

类型2) 训练时空费用大。支持向量机和边框回归训练都需要从每幅图像的每个区域推荐提取特征并写入硬盘, 过程非常耗时, 同时需要很大的存储。

出3) 目标检测速度慢。在测试阶段, 仍然需要从每幅测试图像的每个区域推荐提取特征。

不难看出, R-CNN 的速度慢, 主要是因为对每个区域推荐分别执行卷积网络的前向过程, 没有考虑共享计算。空间金字塔池化网络 (SPPNet) 可以利用共享计算对 R-CNN 加速^[112], 共享的方法是先对整幅输入图像计算一个共享卷积特征图, 再通过空间金字塔池化从这个特征图提取一个固定长度的特征向量, 用来进行每个区域推荐分类。与 R-CNN 相比, SPPNet 的测试时间可加速 10~100 倍, 训练时间可缩短到原来的 1/3。然而, SPPNet 的训练同样是一个多阶段过程, 也需要把提取的特征写入硬盘。SPPNet 的另一个问题是在目标检测任务的微调过程中不能更新 SPP 层之前的卷积层, 从而限制了其精度的提升。因此, 有人提出了快速区域卷积网络 (Fast R-CNN)^[141], 其优点表现在: ①检测质量更高, ②训练过程统一, ③网络全层更新, ④无须磁盘存储。

输入图像和感兴趣区域 (RoI) 先经过卷积池化处理产生卷积特征图, 再经过 RoI 池化层处理得到固定大小的特征图, 之后被全连接层映射成特征向量。每个 RoI 都有两个输出向量: softmax 概率和边框回归偏移 (bounding-box regression offset)。该结构用多任务损失进行端到端训练。

Fast R-CNN 的结构如图 8.2 所示。输入是一幅完整图像和多个区域推荐, 经过几个卷积和最大池化层的处理, 产生一个共享卷积特征图, 用来为每个区域推荐的 RoI 通过最大池化提取一个固定长度的特征向量。这些特征向量输入到一系列全连接层后, 又分化成两个兄弟输出层: 一个输出 K 个目标类别和 1 个背景类别的 softmax 概率估计; 另一个输出 K 个目标类别的精细边框位置, 每个位置用 4 个实数值表示。注意, 每个目标类别都有一个边框回归器, 如果有 K 个目标类别, 就有 K 个边框回归器。

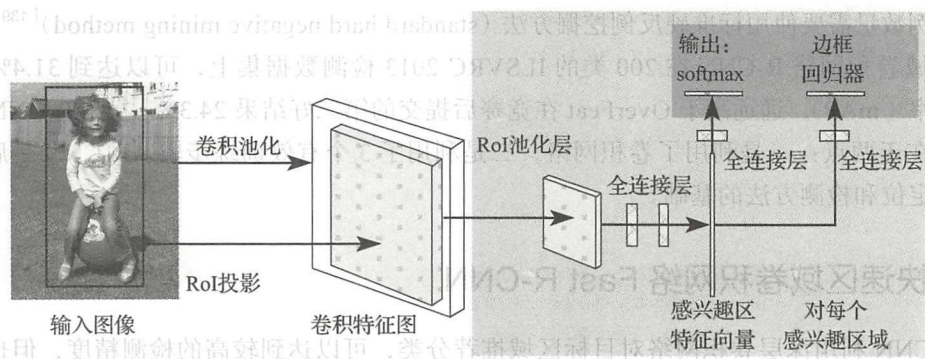


图 8.2 Fast R-CNN 的结构

在 Fast R-CNN 的结构中, RoI 是指卷积特征图中的一个矩形窗口, 可定义为四元组 (r, c, h, w) , 其中 (r, c) 是左上角坐标, (h, w) 是高度和宽度。RoI 池化层利用最大池化把任何有效 RoI 内的特征都转变成一个具有固定空间大小 $H \times W$ (例如 7×7) 的小特征图, 其中 H 和 W 是层超参数, 独立于任何特定 RoI。RoI 最大池化的具体实现过程是先将 $h \times w$ 大小的 RoI 窗口划分成 $H \times W$ 个大小约为 $h/H \times w/W$ 的子窗口, 再把每个子窗口的数值最大池化到相应的输出网格单元。注意, 池化是对每个特征图通道独立应用的。RoI 层也可以看作 SPP 层的特例, 但仅有一个金字塔级。

Fast R-CNN 可以用一个预训练过的深层卷积网络来初始化。这种初始化需要经历 3 个变换: ①把最后的最大池化层替换为一个 RoI 池化层, 其中 H 和 W 的设置应与网络的第一个全连接层相容 (例如, 对 VGGNet-16, $H = W = 7$); ②把最后的全连接层和 softmax 替换为两个兄弟层, 分别用来估计每个类别的 softmax 概率和边框位置; ③把网络改为接收两种数据输入, 一种是图像列表, 另一种是 RoI 列表。

Fast R-CNN 在训练过程中还可以利用特征共享的优点来进一步提高效率。一种策略是对随机梯度下降法的迷你块分层采样, 先采样 N 幅图像, 再从每幅图像采样 R/N 个 RoI, 使得来自同幅图像的 RoI 可以在前向和反向过程中共享计算和内存。例如, $N = 2, R = 128$, 训练速度可加快约 64 倍。另一种策略是基于多任务损失函数, 采用单阶段流式微调过程联合训练 softmax 分类器和边框回归器, 而不是在 3 个独立的阶段分别训练 softmax 分类器、支持向量机和边框回归器。有关细节请参阅文献 [141]。

实验结果表明, 在 Fast R-CNN 的框架下训练 VGGNet-16 比 R-CNN 快 9 倍, 测试快 213 倍, 同时可以获得更高的 mAP 值。与 SPPNet 框架相比, Fast R-CNN 训练 VGGNet-16 的速度快 3 倍, 测试快 10 倍, 检测结果也更精确。Fast R-CNN 为发展更快区域卷积网络 (Faster R-CNN) 奠定了思想和方法基础。

8.3 更快区域卷积网络 Faster R-CNN

8.3.1 Faster R-CNN 的模型结构

虽然 Fast R-CNN 利用共享卷积计算的策略可以有效减少目标检测的运行时间, 但却依赖非常耗时的区域推荐算法提供关于目标位置的假设。为了克服区域推荐算法的瓶颈问题, 需要发展 Faster R-CNN^[142], 如图 8.3 所示。Faster R-CNN 采用了区域推荐网络 (Region Proposal

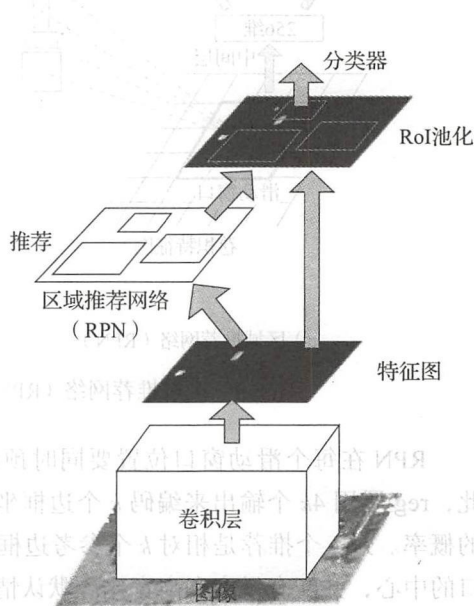


图 8.3 Faster R-CNN 是一个独立统一的目标检测网络, 其中 RPN 模块起到“注意力”的作用。

Network, RPN), 可以与检测网络共享整幅图像的卷积特征, 从而产生几乎无代价的区域推荐。

从图 8.3 不难看出, Faster R-CNN 由两个模块组成。第一个模块是用来产生区域推荐的 RPN, 第二个模块是使用推荐区域的 Fast R-CNN 检测器。整个系统是一个独立统一的目标检测网络, 其中 RPN 模块采用当下流行的“注意力”机制^[143], 告诉 Fast R-CNN 模块应该看什么地方。Faster R-CNN 可以在一个广大的尺度和高宽比范围内检测目标。

RPN 是用一个全卷积网络来实现的^[15], 其输入是一幅(任意大小)图像, 输出是一组带有对象得分的矩形目标推荐, 如图 8.4 所示。引入 RPN 的最终目的是与一个 Fast R-CNN 目标检测网络共享计算, 它们应该共用一组卷积层。RPN 生成区域推荐的办法是在最后一个共享卷积层输出的卷积特征图上滑动一个小网络, 如图 8.4a 所示。这个小网络的输入是卷积特征图上每个滑动位置的 $n \times n$ 空间窗口(一般可取 $n = 3$), 作用是把每个滑动窗口映射到一个低维特征(对 ZFNet 是 256 维, 对 VGGNet-16 是 512 维, 后接 ReLU)。然后, 这个低维特征被输入到两个兄弟全连接层: 一个是边框回归层(reg), 另一个是边框分类层(cls)。由于小网络是按滑动窗口方式操作的, 因此全连接层在所有空间位置都可以共享, 很自然就被实现为一个 $n \times n$ 卷积层再接两个分别对应于 reg 和 cls 的兄弟 1×1 卷积层。

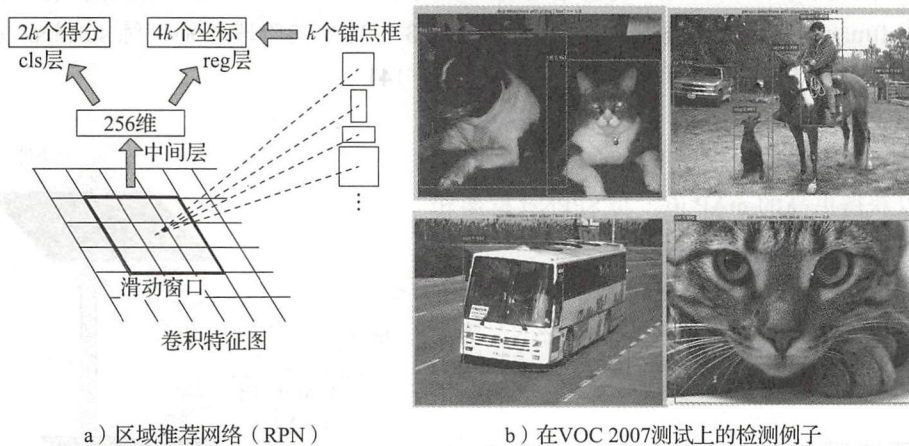


图 8.4 区域推荐网络(RPN)及其在 VOC 2007 测试上的检测例子

RPN 在每个滑动窗口位置要同时预测多个区域推荐, 把最大可能的推荐数记为 k 。因此, reg 层用 $4k$ 个输出来编码 k 个边框坐标, cls 层输出 $2k$ 个分数估计每个推荐是否为对象的概率。这 k 个推荐是相对 k 个参考边框来参数化的, 称为锚点。一个锚点位于当前滑动窗口的中心, 并配有尺度和高宽比。默认情况下采用 3 个尺度(比如 128^2 、 256^2 和 512^2)和 3 个高宽比(比如 $1:1$ 、 $1:2$ 和 $2:1$), 每个滑动位置有 $k = 9$ 个锚点。对于一个大小为 $W \times H$ 的卷积特征图, 总共有 WHk 个锚点。锚点具有平移不变性, 计算锚点关联推荐的函数也具有平移不变性。这就是说, 如果在图像中平移一个对象, 推荐也会跟着平移, 并且在任何位

置都能用相同的函数来预测推荐。此外，锚点的设计也给出了一种解决多尺度（和高宽比）问题的“锚点金字塔”新策略。这种新策略的性价比非常高，能够参考多尺度和多高宽比的锚点框，进行边框分类和回归，但仅依赖单一尺度的图像和特征图，只使用单一尺度的滤波器（即特征图上的滑动窗口）。正是因为这个基于锚点的多尺度设计，才可能像 Fast R-CNN 检测器那样，简单使用在单一尺度图像上计算的卷积特征。多尺度锚点的设计是使用共享特征而不增加额外时间解决尺度问题的关键。

为了训练 PRN，需要给每个锚点分配一个二值类别标签，以说明它是否为对象。被赋予正标签的锚点有两类情况：第一类是与真实边框的交并比最高的锚点，第二类是与任意一个真实边框交并比都高于 0.7 的锚点。注意，一个真实边框有可能把正标签分配给多个锚点。通常第二类条件已经足以决定正样本，仍然采用第一类条件是因为在极少数情况下用第二类条件找不到正样本。一个非正锚点如果与所有真实边框的交并比低于 0.3，会被分配一个负标签。那些既不是正标签也不是负标签的锚点对训练目标是没有贡献的。

构造好正负锚点之后，训练 PRN 的问题就可以转化成最小化一个多任务损失函数。为了方便起见，先定义函数 $\text{smooth}_{L_1}(x)$ ：

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2, & |x| < 1 \\ |x| - 0.5, & \text{其他} \end{cases} \quad (8.1)$$

对于一个锚点的情况，可以把分类损失 L_{cls} 和回归损失 L_{reg} 分别定义为

$$L_{\text{cls}}(p_i, p_i^*) = p_i^* \log p_i + (1 - p_i^*) \log(1 - p_i) \quad (8.2)$$

$$\begin{aligned} L_{\text{reg}}(t_i, t_i^*) &= \text{smooth}_{L_1}(t_i - t_i^*) \\ &= \text{smooth}_{L_1}(x_i - x_i^*) + \text{smooth}_{L_1}(y_i - y_i^*) + \text{smooth}_{L_1}(w_i - w_i^*) + \text{smooth}_{L_1}(h_i - h_i^*) \end{aligned} \quad (8.3)$$

其中， i 是一个迷你批中锚点索引， p_i 是锚点 i 被预测为目标的概率。正锚点的真实标签 p_i^* 是 1，负锚点是 0。 $t_i = (t_{i,x}, t_{i,y}, t_{i,w}, t_{i,h})$ 是一个向量，表示预测边框的 4 个参数化坐标。 $t_i^* = (t_{i,x}^*, t_{i,y}^*, t_{i,w}^*, t_{i,h}^*)$ 也是一个向量，表示一个正锚点的真实边框坐标。而且， t_i 和 t_i^* 的各分量还满足下面的一般数学关系：

$$\begin{cases} t_x = (x - x_a)/w_a, t_y = (y - y_a)/h_a, t_w = \log(w/w_a), t_h = \log(h/h_a) \\ t_x^* = (x^* - x_a)/w_a, t_y^* = (y^* - y_a)/h_a, t_w^* = \log(w^*/w_a), t_h^* = \log(h^*/h_a) \end{cases} \quad (8.4)$$

其中， x 、 y 、 w 和 h 代表边框的中心坐标、宽和高。变量 x 、 x_a 、 x^* 分别代表预测边框、锚点边框和真实边框的相应坐标值（变量 y 、 w 、 h 的情况类似）。

对于多个锚点的情况，如果分类层 cls 和回归层 reg 的输出分别由 $\{p_i\}$ 和 $\{t_i\}$ 组成，那么可以把归一化分类损失 NCLS、归一化回归损失 NREG 和总体加权损失函数 L 分别定义为

$$\text{NCLS}(\{p_i\}, \{t_i\}) = \frac{1}{N_{\text{cls}}} \sum_i L_{\text{cls}}(p_i, p_i^*) \quad (8.5)$$

$$\text{NREG}(\{p_i\}, \{t_i\}) = \frac{1}{N_{\text{reg}}} \sum_i p_i^* L_{\text{reg}}(t_i, t_i^*) \quad (8.6)$$

$$L(\{p_i\}, \{t_i\}) = \text{NCLS} + \lambda \cdot \text{NREG} = \frac{1}{N_{\text{cls}}} \sum_i L_{\text{cls}}(p_i, p_i^*) + \lambda \frac{1}{N_{\text{reg}}} \sum_i p_i^* L_{\text{reg}}(t_i, t_i^*) \quad (8.7)$$

其中, N_{cls} 和 N_{reg} 分别用来归一化分类损失和回归损失。参数 λ 是用来在 NCLS 和 NREG 之间加权平衡的。在具体实现时, N_{cls} 可以取为迷你批的大小 (比如 $N_{\text{cls}} = 256$), N_{reg} 可以取为锚点位置的数目 (比如 N_{reg} 约取为 2400)。 $p_i^* L_{\text{reg}}$ 项表示回归损失仅对正锚点 ($p_i^* = 1$) 才被激活, 否则不起作用。在默认情况下, 设置 $\lambda = 10$, 大致可使 NCLS 和 NREG 之间的平衡保持在比较合理的水平。

Faster R-CNN 为了在 RPN 和 Fast R-CNN 之间共享特征, 采用了一种实用的交替优化训练算法, 包括 4 个步骤。第 1 步是用 ImageNet 预训练一个模型去初始化 RPN, 并对 RPN 进行端到端的区域推荐任务微调。第 2 步是用 ImageNet 预训练一个模型去初始化 Fast R-CNN 检测网络, 并利用第 1 步的 RPN 对这个检测网络独立训练, 注意此时 RPN 和 Fast R-CNN 还没有共享卷积特征。第 3 步是采用 Fast R-CNN 检测网络初始化 RPN 训练, 但是固定共享卷积层, 只微调 RPN 的独有层, 现在这两个网络才开始共享卷积层。第 4 步保持共享卷积层不变, 只微调 Fast R-CNN 的独有层。

8.3.2 Faster R-CNN 的 TensorFlow 代码实现及说明

作为一种目标检测的著名网络框架, Faster R-CNN 有很多不同的具体实现或改进方式。这里将介绍一个在 Linux 系统下用 TensorFlow 实现的版本, 下载地址为 https://github.com/smallcorgi/Faster-RCNN_TF, 其中的基础网络部分采用 VGGNet 结构。把程序下载解压后, 主目录为 Faster-RCNN_TF-master, 其中包括 4 个文件夹 data、experiments、tools、lib, 以及 3 个文件 gitignore、LICENSE 和 readme.md。这个程序的规模相对庞大, 表 8.1 对有关文件的功能进行汇总和描述。

表 8.1 Faster R-CNN 程序的文件及其功能

文件夹	子文件夹	关键文件	功能
tools		_init_paths.py	为 Fast R-CNN 设置路径
		demo.py	利用预先计算好的目标推荐检测图像中的目标类别
		train_net.py	训练一个 Fast R-CNN 网络在一个感兴趣区域的数据集
		test_net.py	在一个图像数据集上测试 Fast R-CNN 网络
lib	datasets	_init_.py	初始化, 添加 matlab_r2013b 路径
		coco.py	数据集 COCO
		ds_utils.py	边框坐标转换, 确定 box 的有效性, 去除小 box
		factory.py	使得可以通过名字更轻松获取 imdbs
		imagenet3d.py	数据集 ImageNet
		imdb.py	数据集的处理, 定义 imdb 对象
		kitti.py	数据集 kitti

(续)

文件夹	子文件夹	关键文件	功能
lib	datasets	nissan.py	数据集 nissan
		Nthu.py	数据集 nthu
		pascal_voc.py	继承 imdb, 扩充功能用于专门处理 VOC 数据集
		VOC_eval.py	解析 VOC xml 文件
	fast-rcnn	_init_.py	导入 config、test、train
		bbox_transform.py	bbox 变换
		config.py	Fast R-CNN 配置系统, 设置各种参数
		nms_wrapper.py	nms
		train.py	训练一个 Fast R-CNN 网络
		test.py	测试一个 Fast R-CNN 网络
	gt_data_layer	_init_.py	空内容
		layer.py	GtDataLayer 执行一个 Caffe Python 层, 以训练数据层
		minibatch.py	为了训练一个 Fast R-CNN 网络, 计算小批量 blobs
		roidb.py	通过增加一串元数据, 将一个 roidb 变换成一个可以训练的 roidb
	networks	_init_.py	导入 VGGnet_train、VGGnet_test、factory
		factory.py	使得可以通过名字比较轻松地获取网络
		network.py	定义网络各种层
		VGGnet_train.py	定义 VGG 网络结构, VGGnet_train
		VGGnet_test.py	用于测试的 VGG 网络
	nms	_init_.py, cpu_nms.pyx, gpu_nms.hpp, gpu_nms.pyx, nms_kernel.cu, py_cpu_nms.py	非极大值抑制
	roi_data_layer	layer.py	RoIDataLayer 类, 用于训练 Fast R-CNN 网络的数据层
		minibatch.py	为了训练一个 Fast R-CNN 网络, 计算小批量 blobs
		roidb.py	增加元数据, 将 roidb 变换成可以训练的 roidb
	roi_pooling_layer	_init_.py, roi_pooling_op.cc, roi_pooling_op.py, roi_pooling_op_gpu.cu.cc, roi_pooling_op_gpu.h, roi_pooling_op_grad.py, roi_pooling_op_test.py, work_sharder.h.	ROI 池化层
	rpn_msr	anchor_target_layer.py, anchor_target_layer_tf.py	为 ground-truth 目标分配锚点, 确定正负标签
		generate_anchors.py	生成不同尺度、长宽比的锚点
		generate.py	对 imdb 格式的图像生成 RPN 推荐区域
		proposal_layer.py, proposal_layer_tf.py	利用 Bbox 变换将锚点变换成推荐区域
		proposal_target_layer_tf.py	产生推荐分类标签和 Bbox 回归目标

(续)

文件夹	子文件夹	关键文件	功能
lib	utils	bbox.pyx	box 和查询 box 之间的重合度
		blob.py	准备图像用于 blob
		boxes_grid.py	返回图像网格上的 box
		nms.py, Nms.pyx	非极大值抑制
		timer.py	计时器
	setup.py		环境配置设置
	makefile, make.sh		
experiments	cfgs	faster_rcnn_alt_opt.yml, faster_rcnn_end2end.yml, kitti_rcnn.yml	YAML 语言写的配置文件
	scripts	faster_rcnn_end2end.sh	Python 脚本命令，用于 Faster R-CNN 训练和测试
data	demo		存放演示图像

根据表 8.1，不难看出 Faster R-CNN 程序包含以下文件结构：

- 1) data 文件夹中存放实验图像；
- 2) experiments 文件夹里面包含两个文件夹：cfgs、scripts；
- 3) tools 文件夹里面包含 4 个文件：_init_paths.py、demo.py、train_net.py 和 test_net.py；
- 4) lib 文件夹里面包含 9 个文件夹：datasets、fast_rcnn、gt_data_layer、networks、nms、roi_data_layer、roi_pooling_layer、rpn_msr、utils，以及 3 个文件 make.sh、Makefile、setup.py。

表 8.2 总结了 Faster R-CNN 程序中最关键的 7 个文件，分别是 train_net.py、train.py、pascal_voc.py、networks.py、generate_anchors.py、anchor_target_layer_tf.py、proposal_layer_tf.py。这些文件的详细说明如下。

表 8.2 Faster R-CNN 程序的关键文件及其功能

关键文件	功能
train_net.py	开始网络训练
train.py	网络的训练过程计算
pascal_voc.py	数据集 pascal_voc 的处理
networks.py	定义网络的各个层以及操作
generate_anchors.py	生成不同尺度和长宽比的锚点
anchor_target_layer_tf.py	通过与 GT bbox 回归生成锚点的标签和训练目标
proposal_layer_tf.py	生成推荐区域

1. train_net.py 文件的代码实现及说明

```
# 在感兴趣区数据库 roidb (Region of Interest database) 上训练 Fast R-CNN
import _init_paths
from fast_rcnn.train import get_training_roidb, train_net
```


#……省略的导入语句，请参考源代码

```
def parse_args():    # 解析输入参数
    parser = argparse.ArgumentParser(description='Train a Fast R-CNN network')
    parser.add_argument('--device', dest='device', help='device to use', default='cpu', type=str)
    parser.add_argument('--device_id', dest='device_id', help='device id to use', default=0, type=int)
    parser.add_argument('--solver', dest='solver', help='solver prototxt', default=None, type=str)
    parser.add_argument('--iters', dest='max_iters', help='number of iterations to train', default=70000, type=int)
    parser.add_argument('--weights', dest='pretrained_model',
                        help='initialize with pretrained model weights', default=None, type=str)
    parser.add_argument('--cfg', dest='cfg_file', help='optional config file', default=None, type=str)
    parser.add_argument('--imdb', dest='imdb_name', help='dataset to train on', default='kitti_train', type=str)
    parser.add_argument('--rand', dest='randomize', help='randomize (do not use a fixed seed)', action='store_true')
    parser.add_argument('--network', dest='network_name', help='name of the network', default=None, type=str)
    parser.add_argument('--set', dest='set_cfgs', help='set config keys', default=None, nargs=argparse.REMAINDER)
    if len(sys.argv) == 1:    # 如果 sys.argv 长度为 1，则说明没有参数传入，系统退出
        parser.print_help()
        sys.exit(1)
    args = parser.parse_args()
    return args

if __name__ == '__main__':
    args = parse_args()
    print('Called with args:')
    print(args)
    if args.cfg_file is not None: # 如果还有其他配置文件，就加载
        cfg_from_file(args.cfg_file)
    if args.set_cfgs is not None:
        cfg_from_list(args.set_cfgs)
    print('Using config:')
    pprint.pprint(cfg)    # pprint 是一种标准、格式化输出方式；输出格式整齐，便于阅读
    if not args.randomize:
        np.random.seed(cfg.RNG_SEED) # 固定随机种子以确保可重复性
    imdb = get_imdb(args.imdb_name)    # 通过数据集名字 imdb_name 获取数据集对应的类的对象
    print 'Loaded dataset '{:s}' for training'.format(imdb.name)
    # imdb_name 默认是“voc_2007_trainval”
    roidb = get_training_roidb(imdb)
    # fast_rcnn 下的 train.py 文件中的函数，返回感兴趣区域数据集 roidb
    output_dir = get_output_dir(imdb, None)    # 输出全路径并保存
    print 'Output will be saved to '{:s}'.format(output_dir)
    device_name = '{}/{:d}'.format(args.device, args.device_id)
    network = get_network(args.network_name)    # 根据网络名称得到其结构和参数
    print 'Use network '{:s}' in training'.format(args.network_name)
    train_net(network, imdb, roidb, output_dir,
```

```
retrained_model=args.pretrained_model, max_iters=args.max_iters)
# fast_rcnn下的train.py文件中的函数，进行网络训练
```

2. train.py 文件的代码实现及说明

```
# 训练一个 Fast R-CNN 网络
from fast_rcnn.config import cfg
import gt_data_layer.roidb as gdl_roidb
#.....省略的导入语句，请参考源代码

class SolverWrapper(object):    # 用来对学到的边框回归权值去归一化
    def __init__(self, sess, saver, network, imdb, roidb, output_dir,
        pretrained_model=None): # 初始化
        self.net = network
        self.imdb = imdb
        self.roidb = roidb
        self.output_dir = output_dir
        self.pretrained_model = pretrained_model
        print 'Computing bounding-box regression targets...'
        if cfg.TRAIN.BBOX_REG:    # 计算预测边框与真实边框的统计信息
            self.bbox_means, self.bbox_stds = rdl_roidb.add_bbox_regression_targets(roidb)
        self.saver = saver

    def snapshot(self, sess, iter): # 在对学习到的边框回归权值进行去归一化后进行快照保存
        net = self.net
        if cfg.TRAIN.BBOX_REG and net.layers.has_key('bbox_pred'):
            with tf.variable_scope('bbox_pred', reuse=True):
                weights = tf.get_variable("weights")
                biases = tf.get_variable("biases")
                orig_0 = weights.eval()
                orig_1 = biases.eval()
                weights_shape = weights.get_shape().as_list()
                sess.run(net.bbox_weights_assign, feed_dict={net.bbox_weights: orig_0 * \
                    np.tile(self.bbox_stds, (weights_shape[0], 1))})
                sess.run(net.bbox_bias_assign, feed_dict={net.bbox_biases: orig_1 * \
                    self.bbox_stds + self.bbox_means})
        if not os.path.exists(self.output_dir):
            os.makedirs(self.output_dir)
        infix = ('_' + cfg.TRAIN.SNAPSHOT_INFIX
            if cfg.TRAIN.SNAPSHOT_INFIX != '' else '')
        filename = (cfg.TRAIN.SNAPSHOT_PREFIX + infix + '_iter_{:d}'.format(iter+1) + \
            '.ckpt')
        filename = os.path.join(self.output_dir, filename)
        self.saver.save(sess, filename)
        print 'Wrote snapshot to: {:s}'.format(filename)
        if cfg.TRAIN.BBOX_REG and net.layers.has_key('bbox_pred'):
            with tf.variable_scope('bbox_pred', reuse=True):
                sess.run(net.bbox_weights_assign, feed_dict={net.bbox_weights: orig_0})
                sess.run(net.bbox_bias_assign, feed_dict={net.bbox_biases: orig_1})

    def _modified_smooth_l1(self, sigma, bbox_pred, bbox_targets, bbox_inside_weights,
        bbox_outside_weights):
        """ ResultLoss = outside_weights * SmoothL1(inside_weights * (bbox_pred - \
            bbox_targets))
        SmoothL1(x) = 0.5 * (sigma * x)^2,    if |x| < 1 / sigma^2
                |x| - 0.5 / sigma^2,    otherwise """ # 计算 smooth_L1 损失
```

```

sigma2 = sigma * sigma
inside_mul = tf.multiply(bbox_inside_weights, tf.subtract(bbox_pred,
bbox_targets))
smooth_l1_sign = tf.cast(tf.less(tf.abs(inside_mul), 1.0 / sigma2),
tf.float32)
smooth_l1_option1 = tf.multiply(tf.multiply(inside_mul, inside_mul),
0.5 * sigma2)
smooth_l1_option2 = tf.subtract(tf.abs(inside_mul), 0.5 / sigma2)
smooth_l1_result = tf.add(tf.multiply(smooth_l1_option1, smooth_l1_sign)
tf.multiply(smooth_l1_option2, tf.abs(tf.subtract(smooth_l1_sign, 1.0))))
outside_mul = tf.multiply(bbox_outside_weights, smooth_l1_result)
return outside_mul

def train_model(self, sess, max_iters):          # 训练网络模型的循环过程
data_layer = get_data_layer(self.roidb, self.imdb.num_classes)
rpn_cls_score = tf.reshape(self.net.get_output('rpn_cls_score_reshape'),
[-1,2])
rpn_label = tf.reshape(self.net.get_output('rpn-data')[0],[-1])
# 把包含目标的锚点标签设置为1
rpn_cls_score = tf.reshape(tf.gather(rpn_cls_score,
tf.where(tf.not_equal(rpn_label,-1))),[-1,2])
rpn_label = tf.reshape(tf.gather(rpn_label,tf.where(tf.not_equal(rpn_label,
-1))),[-1])
rpn_cross_entropy = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits\
(logits=rpn_cls_score, labels=rpn_label))
rpn_bbox_pred = self.net.get_output('rpn_bbox_pred') # 预测边框的偏移量
rpn_bbox_targets = tf.transpose(self.net.get_output('rpn-data')[1],\
[0,2,3,1])
rpn_bbox_inside_weights = tf.transpose(self.net.get_output('rpn-data')[2],\
[0,2,3,1])
rpn_bbox_outside_weights = tf.transpose(self.net.get_output('rpn-data')\
[3],[0,2,3,1])
rpn_smooth_l1 = self._modified_smooth_l1(3.0, rpn_bbox_pred,
rpn_bbox_targets, rpn_bbox_inside_weights, rpn_bbox_outside_weights)
rpn_loss_box = tf.reduce_mean(tf.reduce_sum(rpn_smooth_l1,
reduction_indices=[1, 2, 3]))
cls_score = self.net.get_output('cls_score') # R-CNN 检测目标分类损失
label = tf.reshape(self.net.get_output('roi-data')[1],[-1])
cross_entropy = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits\
(logits=cls_score, labels=label))
bbox_pred = self.net.get_output('bbox_pred')
bbox_targets = self.net.get_output('roi-data')[2]
bbox_inside_weights = self.net.get_output('roi-data')[3]
bbox_outside_weights = self.net.get_output('roi-data')[4]
smooth_l1 = self._modified_smooth_l1(1.0, bbox_pred, bbox_targets,\
bbox_inside_weights, bbox_outside_weights) # 计算 smooth_L1 损失
loss_box = tf.reduce_mean(tf.reduce_sum(smooth_l1, reduction_indices=[1]))
loss = cross_entropy + loss_box + rpn_cross_entropy + rpn_loss_box
# 最后的总损失
global_step = tf.Variable(0, trainable=False) # optimizer and learning rate
lr = tf.train.exponential_decay(cfg.TRAIN.LEARNING_RATE, global_step,\
cfg.TRAIN.STEPSIZE, 0.1, staircase=True)
momentum = cfg.TRAIN.MOMENTUM # 动态系数为 0.9 的梯度下降法
train_op = tf.train.MomentumOptimizer(lr, momentum).minimize(loss,\

```



```

global_step=global_step)
sess.run(tf.global_variables_initializer()) # 全局变量初始化
if self.pretrained_model is not None:      # 如果有预训练模型，则加载
    print ('Loading pretrained model '\
           'weights from {s}').format(self.pretrained_model)
    self.net.load(self.pretrained_model, sess, self.saver, True)
last_snapshot_iter = -1
timer = Timer() # 记录当前时间
for iter in range(max_iters):              # 按最大次数循环
    blobs = data_layer.forward()            # 每次获取一个 batch 的数据信息
    feed_dict={self.net.data: blobs['data'],\
               self.net.im_info: blobs['im_info'], self.net.keep_prob: 0.5, \
               self.net.gt_boxes: blobs['gt_boxes']} run_options = None
    run_metadata = None
    if cfg.TRAIN.DEBUG_TIMELINE:
        run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
        run_metadata = tf.RunMetadata()
    rpn_loss_cls_value, rpn_loss_box_value, loss_cls_value, loss_box_value,
    _ =sess.run([rpn_cross_entropy, rpn_loss_box, cross_entropy, loss_box,\
                 train_op], feed_dict=feed_dict, options=run_options,\
                 run_metadata=run_metadata) # 计算损失值
    if cfg.TRAIN.DEBUG_TIMELINE:
        trace = timeline.Timeline(step_stats=run_metadata.step_stats)
        trace_file = open(str(long(time.time()) * 1000)) + \
            '-train-timeline.ctf.json', 'w')
        trace_file.write(trace.generate_chrome_trace_format(show_memory=\
            False))
        trace_file.close()
    if (iter+1) % (cfg.TRAIN.DISPLAY) == 0:
        print 'iter: %d / %d, total loss: %.4f, rpn_loss_cls: %.4f,\
              rpn_loss_box: %.4f, loss_cls: %.4f, loss_box: %.4f, lr: %f'%\
              (iter+1, max_iters, rpn_loss_cls_value + rpn_loss_box_value + \
               loss_cls_value + loss_box_value, rpn_loss_cls_value, rpn_loss_box_value,\
               loss_cls_value, loss_box_value, lr.eval())
        # 打印迭代次数以及损失值
        print 'speed: {:.3f}s / iter'.format(timer.average_time)
    if (iter+1) % cfg.TRAIN.SNAPSHOT_ITERS == 0:
        last_snapshot_iter = iter
        self.snapshot(sess, iter)
    if last_snapshot_iter != iter:
        self.snapshot(sess, iter)
def get_training_roidb(imdb): # 得到一个训练用的感兴趣区域数据集 roidb
    if cfg.TRAIN.USE_FLIPPED: # 对训练样本进行水平翻转
        print 'Appending horizontally-flipped training examples...'
        imdb.append_flipped_images()
    print 'Preparing training data...'
    if cfg.TRAIN.HAS_RPN:
        if cfg.IS_MULTISCALE:
            gdl_roidb.prepare_roidb(imdb)
        else:
            rdl_roidb.prepare_roidb(imdb)
    else:
        rdl_roidb.prepare_roidb(imdb)

```

```

print 'done'
return imdb.roidb

def filter_roidb(roidb):    # 对 roidb 数据进行筛选, 去除没有有效 RoI 的数据
def is_valid(entry):    # 有效图像至少有一个前景 RoI 或者一个背景 RoI
    overlaps = entry['max_overlaps']    # 与标准框的重叠率
    fg_inds = np.where(overlaps >= cfg.TRAIN.FG_THRESH)[0]
    # 大于设定阈值, 则认为是前景目标
    bg_inds = np.where((overlaps < cfg.TRAIN.BG_THRESH_HI) &
                       (overlaps >= cfg.TRAIN.BG_THRESH_LO))[0]
    valid = len(fg_inds) > 0 or len(bg_inds) > 0
    # 有效图像必须至少包含一个前景或者背景
    return valid

num = len(roidb)
# roidb 是一个字典列表, roidb[im_ind] 代表 im_ind 索引图片所包含的 RoI 信息
filtered_roidb = [entry for entry in roidb if is_valid(entry)]
# 记录筛选后的 roidb
num_after = len(filtered_roidb)
print 'Filtered {} roidb entries: {} -> {}'.format(num - num_after, num, \
num_after)
return filtered_roidb

def train_net(network, imdb, roidb, output_dir, pretrained_model=None, \
max_iters=40000):
    roidb = filter_roidb(roidb)
    saver = tf.train.Saver(max_to_keep=100)    # 对参数进行保存, 100 次迭代更新一次
    with tf.Session(config=tf.ConfigProto(allow_soft_placement=True)) as sess:
        sw = SolverWrapper(sess, saver, network, imdb, roidb, output_dir, \
pretrained_model=pretrained_model)

```

3. pascal_voc.py 文件的代码实现及说明

#……省略的导入语句, 请参考源代码

```

class pascal_voc(imdb):    # pascal_voc 继承 imdb
def __init__(self, image_set, year, devkit_path=None):    # 初始化
    imdb.__init__(self, 'voc_' + year + '_' + image_set)
    self._year = year
    self._image_set = image_set
    self._devkit_path = self._get_default_path() if devkit_path is None else \
devkit_path
    self._data_path = os.path.join(self._devkit_path, 'VOC' + self._year)
    self._classes = ('__background__', 'aeroplane', 'bicycle', 'bird', 'boat',
'bubble', 'bus', 'car', 'cat', 'chair', 'cow', 'diningtable', 'dog', 'horse',
'motorbike', 'person', 'pottedplant', 'sheep', 'sofa', 'train', 'tvmonitor')
    # 共 21 个类别, 类别数是 nclasses 值
    self._class_to_ind = dict(zip(self.classes, xrange(self.num_classes)))
    # 产生类别索引, 如 {'aeroplane':1}
    self._image_ext = '.jpg'    # 图片格式为 jpg
    self._image_index = self._load_image_set_index()    # 包含对应数据集图像名称的列表
    self._roidb_handler = self.gt_roidb    # 得到 roidb 数据对象
    self._salt = str(uuid4())    # 对于分布式数据, 为每个数据生成唯一的标识符
    self._comp_id = 'comp4'
    self.config = {'cleanup'      : True,    # PASCAL 相关的配置选项
'use_salt'      : True,
'use_diff'      : False,

```

```

        'matlab_eval' : False,
        'rpn_file'    : None,
        'min_size'    : 2)
assert os.path.exists(self._devkit_path), 'VOCdevkit path does not exist: {}'.format(self._devkit_path)
assert os.path.exists(self._data_path), 'Path does not exist: {}'.format(self._data_path)
def image_path_at(self, i): # 返回图像序列中第 i 幅图像的绝对路径
    return self.image_path_from_index(self._image_index[i])
def image_path_from_index(self, index): # 根据图像索引标识符构造其绝对路径
    image_path = os.path.join(self._data_path, 'JPEGImages', \
        index + self._image_ext)
    assert os.path.exists(image_path), 'Path does not exist: {}'.format(image_path)
    return image_path
def _load_image_set_index(self): # 加载数据集文件中的图像索引
    # 文件路径举例: self._devkit_path + /VOCdevkit2007/VOC2007/ImageSets/Main/val.txt
    image_set_file = os.path.join(self._data_path, 'ImageSets', 'Main', \
        self._image_set + '.txt')
    assert os.path.exists(image_set_file), 'Path does not exist: {}'.format(image_set_file)
    with open(image_set_file) as f:
        image_index = [x.strip() for x in f.readlines()]
    return image_index
def _get_default_path(self): # 返回期望安装 PASCAL VOC 的缺省路径
    return os.path.join(cfg.DATA_DIR, 'VOCdevkit' + self._year)
def gt_roidb(self): # 返回真实感兴趣数据库, 用于加速之后的调用
    cache_file = os.path.join(self.cache_path, self.name + '_gt_roidb.pkl')
    if os.path.exists(cache_file):
        with open(cache_file, 'rb') as fid:
            roidb = cPickle.load(fid)
            print '{} gt roidb loaded from {}'.format(self.name, cache_file)
        return roidb
    gt_roidb = [self._load_pascal_annotation(index) for index in \
        self._image_index]
    with open(cache_file, 'wb') as fid:
        cPickle.dump(gt_roidb, fid, cPickle.HIGHEST_PROTOCOL)
    print 'wrote gt roidb to {}'.format(cache_file)
    return gt_roidb
def selective_search_roidb(self): # 返回选择性搜索的感兴趣区数据库, 加速调用
    cache_file = os.path.join(self.cache_path, self.name + \
        '_selective_search_roidb.pkl')
    if os.path.exists(cache_file):
        with open(cache_file, 'rb') as fid:
            roidb = cPickle.load(fid)
            print '{} ss roidb loaded from {}'.format(self.name, cache_file)
        return roidb
    if int(self._year) == 2007 or self._image_set != 'test':
        gt_roidb = self.gt_roidb()
        ss_roidb = self._load_selective_search_roidb(gt_roidb)
        roidb = imdb.merge_roidbs(gt_roidb, ss_roidb)
    else:
        roidb = self._load_selective_search_roidb(None)

```



```

with open(cache_file, 'wb') as fid:
    cPickle.dump(roidb, fid, cPickle.HIGHEST_PROTOCOL)
print 'wrote ss roidb to {}'.format(cache_file)
return roidb

def rpn_roidb(self):
    if int(self._year) == 2007 or self._image_set != 'test':
        gt_roidb = self.gt_roidb()
        rpn_roidb = self._load_rpn_roidb(gt_roidb)
        roidb = imdb.merge_roidbs(gt_roidb, rpn_roidb)
        # 将 RPN 推荐的 RoI 与真实边框合并
    else:
        roidb = self._load_rpn_roidb(None)
    return roidb

def _load_rpn_roidb(self, gt_roidb):
    filename = self.config['rpn_file']
    print 'loading {}'.format(filename)
    assert os.path.exists(filename), \
        'rpn data not found at: {}'.format(filename)
    with open(filename, 'rb') as f:
        box_list = cPickle.load(f)
    return self.create_roidb_from_box_list(box_list, gt_roidb)

def _load_selective_search_roidb(self, gt_roidb):
    filename = os.path.abspath(os.path.join(cfg.DATA_DIR, \
        'selective_search_data', self.name + '.mat'))
    assert os.path.exists(filename), \
        'Selective search data not found at: {}'.format(filename)
    raw_data = sio.loadmat(filename)['boxes'].ravel()
    box_list = []
    for i in xrange(raw_data.shape[0]):
        boxes = raw_data[i][:, (1, 0, 3, 2)] - 1
        keep = ds_utils.unique_boxes(boxes)
        boxes = boxes[keep, :]
        keep = ds_utils.filter_small_boxes(boxes, self.config['min_size'])
        boxes = boxes[keep, :]
        box_list.append(boxes)
    return self.create_roidb_from_box_list(box_list, gt_roidb)

def _load_pascal_annotation(self, index):
    # 根据图像索引从 Annotations 文件夹下找 xml 标注数据
    filename = os.path.join(self._data_path, 'Annotations', index + '.xml')
    tree = ET.parse(filename)
    objs = tree.findall('object')
    if not self.config['use_diff']: # 除去标记为“difficult”的样本
        non_diff_objs = [obj for obj in objs if \
            int(obj.find('difficult').text) == 0]
        objs = non_diff_objs
    num_objs = len(objs)
    boxes = np.zeros((num_objs, 4), dtype=np.uint16) # 初始化边框的坐标数组
    gt_classes = np.zeros((num_objs), dtype=np.int32) # 初始化边框的类别索引数组
    overlaps = np.zeros((num_objs, self.num_classes), dtype=np.float32)
    # 初始化边框的重叠数
    seg_areas = np.zeros((num_objs), dtype=np.float32) # 初始化边框的面积数组
    for ix, obj in enumerate(objs): # 把对象边框导入数据框架
        bbox = obj.find('bndbox')

```

```

        x1 = float(bbox.find('xmin').text) - 1
        y1 = float(bbox.find('ymin').text) - 1
        x2 = float(bbox.find('xmax').text) - 1
        y2 = float(bbox.find('ymax').text) - 1
        cls = self._class_to_ind[obj.find('name').text.lower().strip()]
        boxes[ix, :] = [x1, y1, x2, y2]
        gt_classes[ix] = cls
        overlaps[ix, cls] = 1.0
        # 生成类似 one-hot 的编码，对应的类索引处值为 1，其余为 0
        seg_areas[ix] = (x2 - x1 + 1) * (y2 - y1 + 1) # 计算分割的面积
        overlaps = scipy.sparse.csr_matrix(overlaps) # 对 overlaps 进行压缩
        return {'boxes' : boxes, 'gt_classes': gt_classes, 'gt_overlaps' : overlaps, \
            'flipped' : False, 'seg_areas' : seg_areas}

def _get_comp_id(self):
    comp_id = (self._comp_id + '_' + self._salt if self.config['use_salt']
        else self._comp_id)
    return comp_id

def _get_voc_results_file_template(self):
    # VOCdevkit/results/VOC2007/Main/<comp_id>_det_test_aeroplane.txt
    filename = self._get_comp_id() + '_det_' + self._image_set + '_{:s}.txt'
    path = os.path.join(self._devkit_path, 'results', 'VOC' + self._year, 'Main', \
        filename)
    return path

def _write_voc_results_file(self, all_boxes):
    for cls_ind, cls in enumerate(self.classes):
        if cls == '__background__':
            continue
        print 'Writing {} VOC results file'.format(cls)
        filename = self._get_voc_results_file_template().format(cls)
        with open(filename, 'wt') as f:
            for im_ind, index in enumerate(self.image_index):
                dets = all_boxes[cls_ind][im_ind]
                if dets == []:
                    continue
                for k in xrange(dets.shape[0]): # VOCdevkit 使用从 1 开始的索引
                    f.write('{:s} {:.3f} {:.1f} {:.1f} {:.1f} {:.1f} \
                        n'.format(index, dets[k, -1], dets[k, 0] + 1, \
                            dets[k, 1] + 1, dets[k, 2] + 1, dets[k, 3] + 1))

def _do_python_eval(self, output_dir = 'output'):
    annopath = os.path.join(self._devkit_path, 'VOC' + self._year, 'Annotations', \
        '{:s}.xml')
    imagesetfile = os.path.join(self._devkit_path, 'VOC' + self._year, 'ImageSets', \
        'Main', self._image_set + '.txt')
    cachedir = os.path.join(self._devkit_path, 'annotations_cache')
    aps = []
    use_07_metric = True if int(self._year) < 2010 else False
    # 2010 年修改的 PASCAL VOC 度量
    print 'VOC07 metric? ' + ('Yes' if use_07_metric else 'No')
    if not os.path.isdir(output_dir):
        os.mkdir(output_dir)
    for i, cls in enumerate(self._classes):
        if cls == '__background__':
            continue

```

```

        filename = self._get_voc_results_file_template().format(cls)
        rec, prec, ap = voc_eval(filename, annopath, imagesetfile, cls, cachedir,
                                ovthresh=0.5,
                                use_07_metric=use_07_metric)
        aps += [ap]
        print('AP for {} = {:.4f}'.format(cls, ap))
        with open(os.path.join(output_dir, cls + '_pr.pkl'), 'w') as f:
            cPickle.dump({'rec': rec, 'prec': prec, 'ap': ap}, f)
    print('Mean AP = {:.4f}'.format(np.mean(aps)))
    # 打印一系列结果, 代码省略, 请参考源程序
def _do_matlab_eval(self, output_dir='output'):
    # 与 def _do_python_eval 类似, 这个是 Matlab 接口的评价, 代码省略, 请参考源程序
def evaluate_detections(self, all_boxes, output_dir):
    # 这个函数为测试检测结果用, 代码省略, 具体实现请参考源程序
def competition_mode(self, on):
    # 这个函数也为测试检测结果用, 代码省略, 具体实现请参考源程序
if __name__ == '__main__':
    from datasets.pascal_voc import pascal_voc
    d = pascal_voc('trainval', '2007')
    res = d.roidb
    from IPython import embed; embed()

```

4. network.py 文件的代码实现及说明

#..... 省略的导入语句, 请参考源代码

DEFAULT_PADDING = 'SAME'

def layer(op):

def layer_decorated(self, *args, **kwargs):

如果没有提供操作函数名, 则自动设置一个

name = kwargs.setdefault('name', self.get_unique_name(op.__name__))

op.__name__ 是函数名

if len(self.inputs)==0: # 如果没有输入

raise RuntimeError('No input variables found for layer %s.' % name)

elif len(self.inputs)==1:

layer_input = self.inputs[0] # 直接读取输入数据

else:

layer_input = list(self.inputs)

layer_output = op(self, layer_input, *args, **kwargs)

用操作函数做运算 (卷积、池化等)

self.layers[name] = layer_output

self.feed(layer_output) # 当前层的输出输入到下一层

return self # 返回自己做递归调用

return layer_decorated

class Network(object):

def __init__(self, inputs, trainable=True):

self.inputs = []

self.layers = dict(inputs)

self.trainable = trainable

self.setup()

def setup(self):

raise NotImplementedError('Must be subclassed.')

def load(self, data_path, session, saver, ignore_missing=False):

if data_path.endswith('.ckpt'):


```

        saver.restore(session, data_path)
    else:
        data_dict = np.load(data_path).item()
        for key in data_dict:
            with tf.variable_scope(key, reuse=True):
                for subkey in data_dict[key]:
                    try:
                        var = tf.get_variable(subkey)
                        session.run(var.assign(data_dict[key][subkey]))
                        print "assign pretrain model "+subkey+ " to "+key
                    except ValueError:
                        print "ignore "+key
                        if not ignore_missing:
                            raise

def feed(self, *args):
    assert len(args) != 0
    self.inputs = []
    for layer in args:
        if isinstance(layer, basestring):
            try:
                layer = self.layers[layer] #layers 是一个词典
                print layer
            except KeyError:
                print self.layers.keys()
                raise KeyError('Unknown layer name fed: %s'%layer)
        self.inputs.append(layer) # 将取出的 layer 数据存入 input 列表
    return self

def get_output(self, layer):
    try:
        layer = self.layers[layer] #self.layers 记录的是每一层的输出
    except KeyError:
        print self.layers.keys()
        raise KeyError('Unknown layer name fed: %s'%layer)
    return layer

def get_unique_name(self, prefix): # 得到唯一的名字, prefix 指 conv、max_pool.....
    id = sum(t.startswith(prefix) for t, _ in self.layers.items())+1
    return '%s_%d'%(prefix, id) # 返回的就是层名, 类似于 conv_4

def make_var(self, name, shape, initializer=None, trainable=True): # 制作变量
    return tf.get_variable(name, shape, initializer=initializer, \
        trainable=trainable)

def validate_padding(self, padding):
    assert padding in ('SAME', 'VALID') # 判断 padding 类型是否符合要求
    @layer # 定义卷积层

def conv(self, input, k_h, k_w, c_o, s_h, s_w, name, relu=True, \
    padding=DEFAULT_PADDING, group=1, trainable=True):
    self.validate_padding(padding)
    c_i = input.get_shape()[-1] # 获取 input 形状 [batch, in_height, in_width, \
    in_channels]
    assert c_i%group==0
    assert c_o%group==0
    convolve = lambda i, k: tf.nn.conv2d(i, k, [1, s_h, s_w, 1], padding=padding)
    with tf.variable_scope(name) as scope:
        init_weights = tf.truncated_normal_initializer(0.0, stddev=0.01)

```

```

# 用截断正态初始化权重
init_biases = tf.constant_initializer(0.0) # 把偏置初始化为常数 0.0
kernel = self.make_var('weights', [k_h, k_w, c_i/group, c_o],
    init_weights, trainable)
biases = self.make_var('biases', [c_o], init_biases, trainable)
# 制作偏置变量
if group==1:
    conv = convolve(input, kernel)
else:
    input_groups = tf.split(3, group, input)
    kernel_groups = tf.split(3, group, kernel)
    output_groups = [convolve(i, k) for i, k in zip(input_groups, \
        kernel_groups)]
    conv = tf.concat(3, output_groups)
if ReLU:
    bias = tf.nn.bias_add(conv, biases)
    return tf.nn.relu(bias, name=scope.name)
return tf.nn.bias_add(conv, biases, name=scope.name)
@layer # 定义 ReLU 激活函数
def relu(self, input, name):
    return tf.nn.relu(input, name=name)
@layer # 定义最大池化层
def max_pool(self, input, k_h, k_w, s_h, s_w, name, padding=DEFAULT_PADDING):
    self.validate_padding(padding)
    return tf.nn.max_pool(input, ksize=[1, k_h, k_w, 1], strides=[1, s_h, s_w, 1],
        padding=padding, name=name)
@layer # 定义平均池化层
def avg_pool(self, input, k_h, k_w, s_h, s_w, name, padding=DEFAULT_PADDING):
    self.validate_padding(padding)
    return tf.nn.avg_pool(input, ksize=[1, k_h, k_w, 1], strides=[1, s_h, s_w, 1], \
        padding=padding, name=name)
@layer # 定义感兴趣区域池化层
def roi_pool(self, input, pooled_height, pooled_width, spatial_scale, name):
    if isinstance(input[0], tuple): # 只用第 1 层
        input[0] = input[0][0]
    if isinstance(input[1], tuple):
        input[1] = input[1][0]
    print input
    return roi_pool_op.roi_pool(input[0], input[1], pooled_height, \
        pooled_width, spatial_scale, name=name)[0]
@layer # 区域推荐层
def proposal_layer(self, input, _feat_stride, anchor_scales, cfg_key, name):
    if isinstance(input[0], tuple):
        input[0] = input[0][0]
    return tf.reshape(tf.py_func(proposal_layer_py, [input[0], input[1], input[2], \
        cfg_key, _feat_stride, anchor_scales], [tf.float32]), [-1, 5], name =name)
@layer # 返回每个锚点和锚点的回归值
def anchor_target_layer(self, input, _feat_stride, anchor_scales, name):
    if isinstance(input[0], tuple):
        # input 为 'rpn_cls_score'、'gt_boxes' 和 'im_info'、'data' 信息组成的列表
        input[0] = input[0][0] # input[0] 就是 'rpn_cls_score' 的输出
    with tf.variable_scope(name) as scope:
        rpn_labels, rpn_bbox_targets, rpn_bbox_inside_weights, \

```

```

        rpn_bbox_outside_weights = tf.py_func(anchor_target_layer_py,\
        [input[0],input[1],input[2],input[3],\
        _feat_stride, anchor_scales], [tf.float32,tf.float32,tf.\
        float32,tf.float32])
rpn_labels = tf.convert_to_tensor(tf.cast(rpn_labels,tf.int32), name = 'rpn_labels')
rpn_bbox_targets = tf.convert_to_tensor(rpn_bbox_targets, name = 'rpn_bbox_targets')
rpn_bbox_inside_weights = tf.convert_to_tensor(rpn_bbox_inside_weights ,\
name = 'rpn_bbox_inside_weights')
rpn_bbox_outside_weights = tf.convert_to_tensor(rpn_bbox_outside_weights ,\
name = rpn_bbox_outside_weights')
return rpn_labels, rpn_bbox_targets, rpn_bbox_inside_weights,\
rpn_bbox_outside_weights

@layer
def proposal_target_layer(self, input, classes, name):
    # 对每一个 proposal 生成训练的目标和标签
    if isinstance(input[0], tuple):
        input[0] = input[0][0]
        with tf.variable_scope(name) as scope:
            rois,labels,bbox_targets,bbox_inside_weights,bbox_outside_weights=\
            tf.py_func(proposal_target_layer_py,[input[0],input[1],classes],\
            [tf.float32,tf.float32,tf.float32,tf.float32,tf.float32])
            rois = tf.reshape(rois,[-1,5] , name = 'rois')
            labels = tf.convert_to_tensor(tf.cast(labels,tf.int32), name = 'labels')
            bbox_targets = tf.convert_to_tensor(bbox_targets, name = 'bbox_targets')
            bbox_inside_weights = tf.convert_to_tensor(bbox_inside_weights,\
            name = 'bbox_inside_weights')
            bbox_outside_weights = tf.convert_to_tensor(bbox_outside_weights,\
            name = 'bbox_outside_weights')
            return rois, labels, bbox_targets, bbox_inside_weights, bbox_outside_weights
    @layer
def reshape_layer(self, input, d,name):
    input_shape = tf.shape(input)
    if name == 'rpn_cls_prob_reshape':
        return tf.transpose(tf.reshape(tf.transpose(input,[0,3,1,2]),\
        [input_shape[0], int(d), tf.cast(tf.cast(input_shape[1],tf.float32))\
        /tf.cast(d,tf.float32)*tf.cast(input_shape[3],tf.float32),\
        tf.int32),input_shape[2]]),[0,2,3,1],name=name)
    else:
        return tf.transpose(tf.reshape(tf.transpose(input,[0,3,1,2]),\
        [input_shape[0], int(d), tf.cast(tf.cast(input_shape[1],tf.float32))*\
        (tf.cast(input_shape[3],tf.float32)/tf.cast(d,tf.float32)),tf.int32),\
        input_shape[2]]),[0,2,3,1],name=name)
    @layer
def feature_extrapolating(self, input, scales_base, num_scale_base,\
num_per_octave, name):
    return feature_extrapolating_op.feature_extrapolating(input, scales_base,\
num_scale_base, num_per_octave, name=name)
    @layer
def lrn(self, input, radius, alpha, beta, name, bias=1.0):
    return tf.nn.local_response_normalization(input, depth_radius=radius,\
alpha=alpha, beta=beta, bias=bias, name=name)
    @layer
def concat(self, inputs, axis, name):

```



```

        return tf.concat(concat_dim=axis, values=inputs, name=name)

@layer
def fc(self, input, num_out, name, relu=True, trainable=True):
    with tf.variable_scope(name) as scope:
        if isinstance(input, tuple): # 只用第1层
            input = input[0]
        input_shape = input.get_shape()
        if input_shape.ndims == 4:
            dim = 1
            for d in input_shape[1:].as_list():
                dim *= d
            feed_in = tf.reshape(tf.transpose(input, [0, 3, 1, 2]), [-1, dim])
        else:
            feed_in, dim = (input, int(input_shape[-1]))
        if name == 'bbox_pred':
            init_weights = tf.truncated_normal_initializer(0.0, stddev=0.001)
            init_biases = tf.constant_initializer(0.0)
        else:
            init_weights = tf.truncated_normal_initializer(0.0, stddev=0.01)
            init_biases = tf.constant_initializer(0.0)
        weights = self.make_var('weights', [dim, num_out], init_weights, trainable)
        biases = self.make_var('biases', [num_out], init_biases, trainable)
        op = tf.nn.relu_layer if relu else tf.nn.xw_plus_b
        fc = op(feed_in, weights, biases, name=scope.name)
        return fc

@layer
def softmax(self, input, name):
    input_shape = tf.shape(input)
    if name == 'rpn_cls_prob':
        return tf.reshape(tf.nn.softmax(tf.reshape(input, [-1, input_shape[3]])), \
            [-1, input_shape[1], input_shape[2], input_shape[3]], name=name) # 返回类别概率
    else:
        return tf.nn.softmax(input, name=name)

@layer
def dropout(self, input, keep_prob, name):
    return tf.nn.dropout(input, keep_prob, name=name) # 用 dropout 防止过拟合

```

5. generate_anchors.py 文件的代码实现及说明

```

import numpy as np
def generate_anchors(base_size=16, ratios=[0.5, 1, 2], scales=2*np.arange(3, 6)):
    # 生成锚点 (参考) 窗口
    base_anchor = np.array([1, 1, base_size, base_size]) - 1 # 新建一个数组
    ratio_anchors = _ratio_enum(base_anchor, ratios) # 枚举不同宽高比的锚点
    anchors = np.vstack([_scale_enum(ratio_anchors[i, :], scales) for i in xrange(
        ratio_anchors.shape[0])])
    return anchors
def whctrs(anchor): # 返回锚点的宽、高, x 中心, y 中心
    w = anchor[2] - anchor[0] + 1
    h = anchor[3] - anchor[1] + 1
    x_ctr = anchor[0] + 0.5 * (w - 1)
    y_ctr = anchor[1] + 0.5 * (h - 1)
    return w, h, x_ctr, y_ctr

```

```

def _mkanchors(ws, hs, x_ctr, y_ctr): # 制作锚点
    ws = ws[:, np.newaxis]
    hs = hs[:, np.newaxis]
    anchors = np.hstack((x_ctr - 0.5 * (ws - 1), y_ctr - 0.5 * (hs - 1), \
        x_ctr + 0.5 * (ws - 1), y_ctr + 0.5 * (hs - 1)))
    return anchors

def _ratio_enum(anchor, ratios): # 生成1:2、1:1、2:1的锚点
    w, h, x_ctr, y_ctr = _whctrs(anchor) # 返回宽高和中心坐标, w:16, h:16, x_ctr:7.5, y_
    ctr:7.5
    size = w * h # 计算一个基础大小, w*h=256
    size_ratios = size / ratios
    ws = np.round(np.sqrt(size_ratios))
    hs = np.round(ws * ratios)
    anchors = _mkanchors(ws, hs, x_ctr, y_ctr)
    return anchors

def _scale_enum(anchor, scales): # 扩展锚点的倍数
    w, h, x_ctr, y_ctr = _whctrs(anchor)
    ws = w * scales
    hs = h * scales
    anchors = _mkanchors(ws, hs, x_ctr, y_ctr)
    return anchors

if __name__ == '__main__':
    import time
    t = time.time()
    a = generate_anchors()
    print time.time() - t
    print a
    from IPython import embed; embed()

```

6. anchor_target_layer_tf.py 文件的代码实现及说明

#..... 省略的导入语句, 请参考源代码

DEBUG = False

```

def anchor_target_layer(rpn_cls_score, gt_boxes, im_info, data, \
    _feat_stride = [16,], anchor_scales = [4, 8, 16, 32]): # 锚点目标层
    _anchors = generate_anchors(scales=np.array(anchor_scales)) # 生成锚点
    _num_anchors = _anchors.shape[0] # _num_anchors 等于 9
    if DEBUG:
        print 'anchors:'
        print _anchors
        print 'anchor shapes:'
        print np.hstack((_anchors[:, 2::4] - _anchors[:, 0::4], \
            _anchors[:, 3::4] - _anchors[:, 1::4], ))
        _counts = cfg.EPS
        _sums = np.zeros((1, 4))
        _squared_sums = np.zeros((1, 4))
        _fg_sum = 0
        _bg_sum = 0
        _count = 0
    _allowed_border = 0 # 不允许边框超出图外
    im_info = im_info[0]
    # 对每个 (H, W) 位置 i, 按其中心生成 9 个锚点框
    assert rpn_cls_score.shape[0] == 1
    height, width = rpn_cls_score.shape[1:3]

```

```

if DEBUG:
    print 'AnchorTargetLayer: height', height, 'width', width
    print 'im_size: ({}, {})'.format(im_info[0], im_info[1])
    print 'scale: {}'.format(im_info[2])
    print 'height, width: ({}, {})'.format(height, width)
    print 'rpn: gt_boxes.shape', gt_boxes.shape
    print 'rpn: gt_boxes', gt_boxes
    shift_x = np.arange(0, width) * _feat_stride # 产生 width 个横向偏移值
    shift_y = np.arange(0, height) * _feat_stride # 产生 height 个纵向偏移值
    shift_x, shift_y = np.meshgrid(shift_x, shift_y) # 将坐标向量转换为坐标矩阵
    shifts = np.vstack((shift_x.ravel(), shift_y.ravel(), shift_x.ravel(), \
    shift_y.ravel())).transpose()
    A = _num_anchors
    K = shifts.shape[0]
    all_anchors = (_anchors.reshape((1, A, 4)) + \
    shifts.reshape((1, K, 4)).transpose((1, 0, 2)))
    all_anchors = all_anchors.reshape((K * A, 4))
    total_anchors = int(K * A) # 锚点总数为 K*A 个
    inds_inside = np.where( (all_anchors[:, 0] >= -_allowed_border) & \
    (all_anchors[:, 1] >= -_allowed_border) & (all_anchors[:, 2] < im_info[1] + \
    _allowed_border) & (all_anchors[:, 3] < im_info[0] + _allowed_border) )[0]
    # 只保存图像区域内的锚点, 丢弃外面的
    if DEBUG:
        print 'total_anchors', total_anchors
        print 'inds_inside', len(inds_inside)
    anchors = all_anchors[inds_inside, :]
    if DEBUG:
        print 'anchors.shape', anchors.shape
    labels = np.empty((len(inds_inside), ), dtype=np.float32) # label: 1 is positive,
    0 is negative, -1 is don't care
    labels.fill(-1)
    overlaps = bbox_overlaps( np.ascontiguousarray(anchors, dtype=np.float), \
    np.ascontiguousarray(gt_boxes, dtype=np.float)) # 计算锚点与真实边框的重叠
    argmax_overlaps = overlaps.argmax(axis=1) # 给每个锚点找出重叠最多的真实边框
    max_overlaps = overlaps[np.arange(len(inds_inside)), argmax_overlaps]
    # 计算最大交并比
    gt_argmax_overlaps = overlaps.argmax(axis=0) # 给每个真实边框找出重叠最多的锚点
    gt_max_overlaps = overlaps[gt_argmax_overlaps, np.arange(overlaps.shape[1])]
    # 计算最大交并比
    gt_argmax_overlaps = np.where(overlaps == gt_max_overlaps)[0]

    if not cfg.TRAIN.RPN_CLOBBER_POSITIVES:
        labels[max_overlaps < cfg.TRAIN.RPN_NEGATIVE_OVERLAP] = 0
        # 交并比小于 0.3, 标记为 0
        labels[gt_argmax_overlaps] = 1
        # 与真实边框重叠最多或重叠不低于 0.7 的锚点, 标记为 1
        labels[max_overlaps >= cfg.TRAIN.RPN_POSITIVE_OVERLAP] = 1
    if cfg.TRAIN.RPN_CLOBBER_POSITIVES:
        labels[max_overlaps < cfg.TRAIN.RPN_NEGATIVE_OVERLAP] = 0
        num_fg = int(cfg.TRAIN.RPN_FG_FRACTION * cfg.TRAIN.RPN_BATCHSIZE)
        fg_inds = np.where(labels == 1)[0]
        if len(fg_inds) > num_fg: # 若前景样本太多, 则抽减
            disable_inds = npr.choice(fg_inds, size=(len(fg_inds) - num_fg), \

```



```

        replace=False)
        labels[disable_inds] = -1
    num_bg = cfg.TRAIN.RPN_BATCHSIZE - np.sum(labels == 1)
    bg_inds = np.where(labels == 0)[0]
    if len(bg_inds) > num_bg:      # 若背景样本太多，则抽减
        disable_inds = npr.choice(bg_inds, size=(len(bg_inds) - num_bg),\
            replace=False)
        labels[disable_inds] = -1
    bbox_targets = np.zeros((len(inds_inside), 4), dtype=np.float32)
    bbox_targets = _compute_targets(anchors, gt_boxes[argmax_overlaps, :])
    # 计算每个锚点的回归值
    bbox_inside_weights = np.zeros((len(inds_inside), 4), dtype=np.float32)
    bbox_inside_weights[labels == 1, :] = \
        np.array(cfg.TRAIN.RPN_BBOX_INSIDE_WEIGHTS)
    bbox_outside_weights = np.zeros((len(inds_inside), 4), dtype=np.float32)
    if cfg.TRAIN.RPN_POSITIVE_WEIGHT < 0:
        num_examples = np.sum(labels >= 0)
        positive_weights = np.ones((1, 4)) * 1.0 / num_examples
        negative_weights = np.ones((1, 4)) * 1.0 / num_examples
    else:
        assert ((cfg.TRAIN.RPN_POSITIVE_WEIGHT > 0) & \
            (cfg.TRAIN.RPN_POSITIVE_WEIGHT < 1))
        positive_weights = (cfg.TRAIN.RPN_POSITIVE_WEIGHT / np.sum(labels == 1))
        negative_weights = ((1.0 - cfg.TRAIN.RPN_POSITIVE_WEIGHT) / np.sum(labels == 0))
    bbox_outside_weights[labels == 1, :] = positive_weights
    bbox_outside_weights[labels == 0, :] = negative_weights
    if DEBUG:
        _sums += bbox_targets[labels == 1, :].sum(axis=0)
        _squared_sums += (bbox_targets[labels == 1, :] ** 2).sum(axis=0)
        _counts += np.sum(labels == 1)
        means = _sums / _counts
        stds = np.sqrt(_squared_sums / _counts - means ** 2)
        print 'means:'
        print means
        print 'stdevs:'
        print stds
    labels = _unmap(labels, total_anchors, inds_inside, fill=-1)
    bbox_targets = _unmap(bbox_targets, total_anchors, inds_inside, fill=0)
    bbox_inside_weights = _unmap(bbox_inside_weights, total_anchors,\
        inds_inside, fill=0)
    bbox_outside_weights = _unmap(bbox_outside_weights, total_anchors,\
        inds_inside, fill=0)
    if DEBUG:
        print 'rpn: max max_overlap', np.max(max_overlaps)
        print 'rpn: num_positive', np.sum(labels == 1)
        print 'rpn: num_negative', np.sum(labels == 0)
        _fg_sum += np.sum(labels == 1)
        _bg_sum += np.sum(labels == 0)
        _count += 1
        print 'rpn: num_positive avg', _fg_sum / _count
        print 'rpn: num_negative avg', _bg_sum / _count
    labels = labels.reshape((1, height, width, A)).transpose(0, 3, 1, 2)

```

```

labels = labels.reshape((1, 1, A * height, width))
rpn_labels = labels
bbox_targets = bbox_targets \
.reshape((1, height, width, A * 4)).transpose(0, 3, 1, 2)
rpn_bbox_targets = bbox_targets
bbox_inside_weights = bbox_inside_weights \
.reshape((1, height, width, A * 4)).transpose(0, 3, 1, 2)
rpn_bbox_inside_weights = bbox_inside_weights
bbox_outside_weights = bbox_outside_weights \
.reshape((1, height, width, A * 4)).transpose(0, 3, 1, 2)
rpn_bbox_outside_weights = bbox_outside_weights
return rpn_labels, rpn_bbox_targets, rpn_bbox_inside_weights, \
rpn_bbox_outside_weights
def _unmap(data, count, inds, fill=0):
    if len(data.shape) == 1:
        ret = np.empty((count, ), dtype=np.float32)
        ret.fill(fill)
        ret[inds] = data
    else:
        ret = np.empty((count, ) + data.shape[1:], dtype=np.float32)
        ret.fill(fill)
        ret[inds, :] = data
    return ret
def _compute_targets(ex_rois, gt_rois):
    assert ex_rois.shape[0] == gt_rois.shape[0] # 锚点与最大重叠真实边框应相同
    assert ex_rois.shape[1] == 4                # 锚点左上角与右下角坐标有4个元素
    assert gt_rois.shape[1] == 5                # 真实边框的标签位
    return bbox_transform(ex_rois, gt_rois[:, :4]).astype(np.float32, copy=False)
# 返回锚点回归值数组

```

7. proposal_layer_tf.py 文件的代码实现及说明

```

DEBUG = False
def proposal_layer(rpn_cls_prob_reshape, rpn_bbox_pred, im_info, \
cfg_key, feat_stride = [16, ], anchor_scales = [8, 16, 32]): # 输出估计的目标边框
    _anchors = generate_anchors(scales=np.array(anchor_scales)) # 生成9个锚点
    _num_anchors = _anchors.shape[0]
    rpn_cls_prob_reshape = np.transpose(rpn_cls_prob_reshape, [0, 3, 1, 2]) # RPN 类别概率
    rpn_bbox_pred = np.transpose(rpn_bbox_pred, [0, 3, 1, 2]) # RPN 回归的边界框
    im_info = im_info[0]
    assert rpn_cls_prob_reshape.shape[0] == 1, 'Only single item batches are supported'
    pre_nms_topN = cfg[cfg_key].RPN_PRE_NMS_TOP_N # 在 NMS 之前保留的高分边框数目
    post_nms_topN = cfg[cfg_key].RPN_POST_NMS_TOP_N # 在 NMS 之后保留的高分边框数目
    nms_thresh = cfg[cfg_key].RPN_NMS_THRESH # NMS 阈值
    min_size = cfg[cfg_key].RPN_MIN_SIZE # 推荐区域的最小尺寸
    scores = rpn_cls_prob_reshape[:, _num_anchors:, :, :]
    bbox_deltas = rpn_bbox_pred
    if DEBUG:
        print 'im_size: ({} , {} )'.format(im_info[0], im_info[1])
        print 'scale: {}'.format(im_info[2])
    height, width = scores.shape[-2:] # 特征图的高和宽
    if DEBUG:

```

```

    print 'score map size: {}'.format(scores.shape)
    shift_x = np.arange(0, width) * _feat_stride      # 产生 width 个横向偏移
    shift_y = np.arange(0, height) * _feat_stride     # 产生 height 个纵向偏移值
    shift_x, shift_y = np.meshgrid(shift_x, shift_y) # 将坐标向量转换为坐标矩阵
    shifts = np.vstack((shift_x.ravel(), shift_y.ravel(), shift_x.ravel(), \
    shift_y.ravel())).transpose()
    A = _num_anchors          # 锚点数为 9
    K = shifts.shape[0]       # K 等于 Width*height
    anchors = _anchors.reshape((1, A, 4)) + \
    shifts.reshape((1, K, 4)).transpose((1, 0, 2))
    anchors = anchors.reshape((K * A, 4))
    bbox_deltas = bbox_deltas.transpose((0, 2, 3, 1)).reshape((-1, 4))
    scores = scores.transpose((0, 2, 3, 1)).reshape((-1, 1))
    proposals = bbox_transform_inv(anchors, bbox_deltas) # 把锚点转化为建议区域
    proposals = clip_boxes(proposals, im_info[:2]) # 裁剪预测框, 使得边框位于图像内
    keep = _filter_boxes(proposals, min_size * im_info[2])
    # 除去宽或者高小于阈值的边框
    proposals = proposals[keep, :] # 保存符合条件的建议得分
    scores = scores[keep]
    order = scores.ravel().argsort()[::-1] # 按索引值排序, [::-1] 是反序排列
    if pre_nms_topN > 0:
        order = order[:pre_nms_topN] # 取出排在最前面的 pre_nms_topN 个
    proposals = proposals[order, :] # 保存符合条件的建议得分
    scores = scores[order]
    keep = nms(np.hstack((proposals, scores)), nms_thresh) # 进行非极大值抑制
    if post_nms_topN > 0:
        keep = keep[:post_nms_topN] # 取前 post_nms_topN 个分值比较高的索引
    proposals = proposals[keep, :]
    scores = scores[keep]
    batch_inds = np.zeros((proposals.shape[0], 1), dtype=np.float32) # proposal 索引
    blob = np.hstack((batch_inds, proposals.astype(np.float32, copy=False)))
    # 生成 blob, 有 proposal 索引
    return blob

def _filter_boxes(boxes, min_size): # 除去边长小于阈值的边框
    ws = boxes[:, 2] - boxes[:, 0] + 1
    hs = boxes[:, 3] - boxes[:, 1] + 1
    keep = np.where((ws >= min_size) & (hs >= min_size))[0]
    return keep

```

8.3.3 Faster R-CNN 的图像目标检测案例及演示效果

本节介绍一个利用 Faster R-CNN 在 TensorFlow 框架下进行图像目标检测的案例, 其中用到的 VOC 2007 图像数据集可以根据表 1.2 提供的地址下载。为了加快训练过程, 还需下载经过 ImageNet 预训练的 VGGNet 网络模型 VGG_imagenet.npy, 并存放到 /data/pretrain_model/ 文件夹下。此外, 还可能需要根据方框 8.1 修改文件 lib/datasets/pascal_voc.py 中的数据名称和图像类别等内容。最后, 需要在文件 lib/fast_rcnn/config.py 中, 对学习率、动量值、迷你块大小、IoU 阈值、NMS 阈值等超参数进行重新设置, 比如根据方框 8.2 修改学习率。

方框 8.1 在文件 pascal_voc.py 中的修改内容

```
class pascal_voc(imdb):
    def __init__(self, image_set, year, devkit_path=None):
        imdb.__init__(self, 'voc_' + year + '_' + image_set)
        self._year = year
        self._image_set = image_set
        self._devkit_path = self._get_default_path() if devkit_path is None else devkit_path
        self._data_path = os.path.join(self._devkit_path, 'VOC' + self._year)
        self._classes = ('__background__', 'aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus', 'car', 'cat', 'chair', 'cow', 'diningtable', 'dog', 'horse', 'motorbike', 'person', 'pottedplant', 'sheep', 'sofa', 'train', 'tvmonitor')
        # 根据需要修改图像类别
```

方框 8.2 在文件 config.py 中修改学习率的设置情况

```
#..... 省略部分语句, 请参考源代码
# 学习率
__C.TRAIN.LEARNING_RATE = 0.001
__C.TRAIN.MOMENTUM = 0.9
__C.TRAIN.GAMMA = 0.1
__C.TRAIN.STEPSIZE = 50000
__C.TRAIN.DISPLAY = 10
__C.IS_MULTISCALE = False
#..... 省略部分语句, 请参考源代码
```

Faster R-CNN 网络的训练步骤如下。参照图 8.5, 执行训练命令。如图 8.6 所示, 在训练到 10 次时, Faster R-CNN 网络的全部损失为 0.3780, RPN 网络的分类损失为 0.3743, RPN 网络的回归损失为 0.0036, 检测网络的分类损失和回归损失都为 0, 此时的学习率为 0.001。如图 8.7 所示, 在训练到 70 000 次时结束, 保存结果, 此时 Faster R-CNN 网络总的损失为 0.0112, RPN 网络的分类损失为 0.0073, RPN 网络的回归损失为 0.0039, 检测网络的分类损失和回归损失都为 0, 此时的学习率为 0.0001。

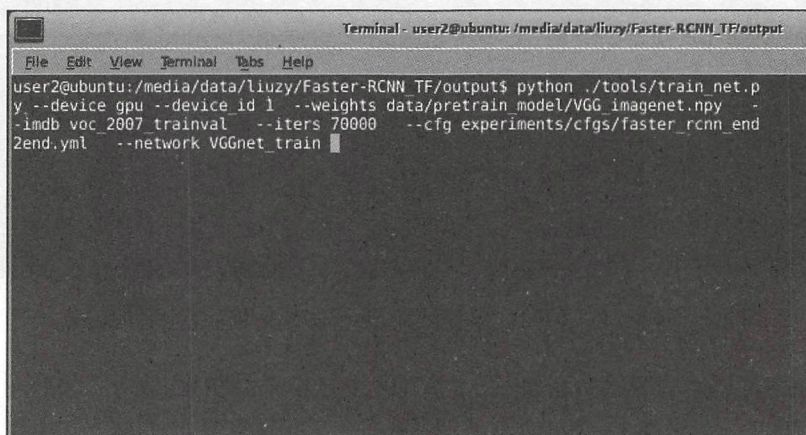
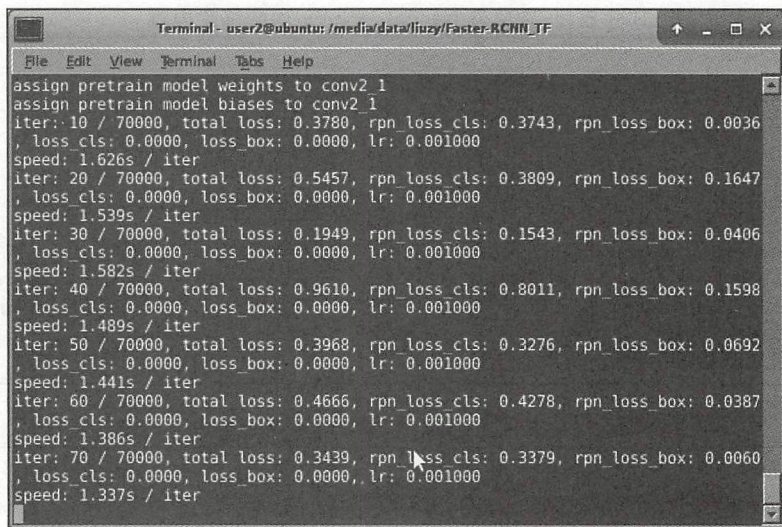


图 8.5 Faster R-CNN 图像目标检测案例程序的训练命令

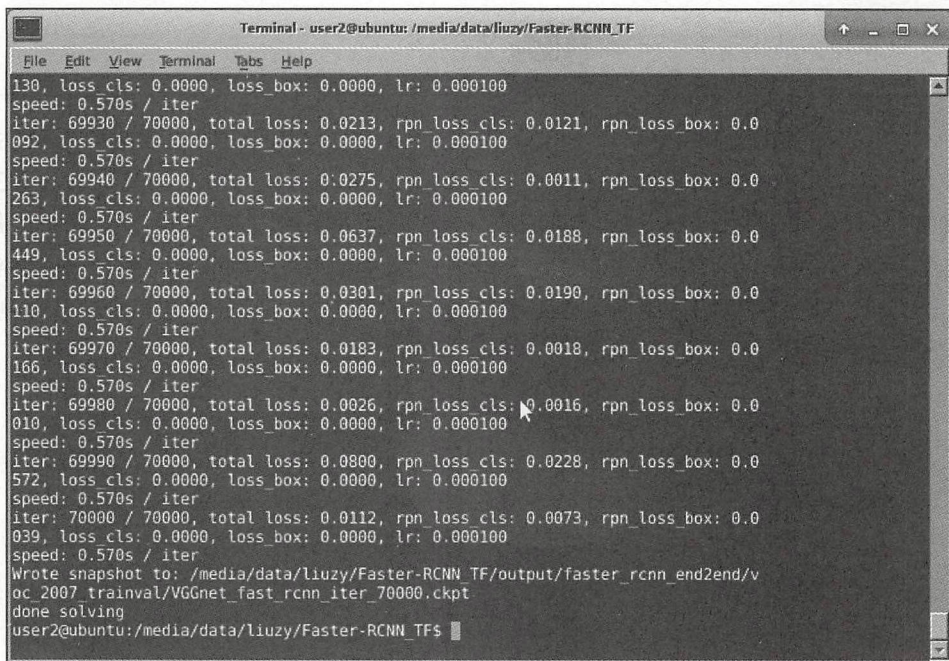


```

Terminal - user2@ubuntu: /media/data/liuzy/Faster-RCNN_TF
File Edit View Terminal Tabs Help
assign pretrain model weights to conv2_1
assign pretrain model biases to conv2_1
iter: 10 / 70000, total loss: 0.3780, rpn_loss_cls: 0.3743, rpn_loss_box: 0.0036
, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.001000
speed: 1.626s / iter
iter: 20 / 70000, total loss: 0.5457, rpn_loss_cls: 0.3809, rpn_loss_box: 0.1647
, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.001000
speed: 1.539s / iter
iter: 30 / 70000, total loss: 0.1949, rpn_loss_cls: 0.1543, rpn_loss_box: 0.0406
, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.001000
speed: 1.582s / iter
iter: 40 / 70000, total loss: 0.9610, rpn_loss_cls: 0.8011, rpn_loss_box: 0.1598
, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.001000
speed: 1.489s / iter
iter: 50 / 70000, total loss: 0.3968, rpn_loss_cls: 0.3276, rpn_loss_box: 0.0692
, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.001000
speed: 1.441s / iter
iter: 60 / 70000, total loss: 0.4666, rpn_loss_cls: 0.4278, rpn_loss_box: 0.0387
, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.001000
speed: 1.386s / iter
iter: 70 / 70000, total loss: 0.3439, rpn_loss_cls: 0.3379, rpn_loss_box: 0.0060
, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.001000
speed: 1.337s / iter

```

图 8.6 Faster R-CNN 图像目标检测案例程序在训练到 10 ~ 70 次的中间信息



```

Terminal - user2@ubuntu: /media/data/liuzy/Faster-RCNN_TF
File Edit View Terminal Tabs Help
130, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.000100
speed: 0.570s / iter
iter: 69930 / 70000, total loss: 0.0213, rpn_loss_cls: 0.0121, rpn_loss_box: 0.0
092, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.000100
speed: 0.570s / iter
iter: 69940 / 70000, total loss: 0.0275, rpn_loss_cls: 0.0011, rpn_loss_box: 0.0
263, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.000100
speed: 0.570s / iter
iter: 69950 / 70000, total loss: 0.0637, rpn_loss_cls: 0.0188, rpn_loss_box: 0.0
449, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.000100
speed: 0.570s / iter
iter: 69960 / 70000, total loss: 0.0301, rpn_loss_cls: 0.0190, rpn_loss_box: 0.0
110, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.000100
speed: 0.570s / iter
iter: 69970 / 70000, total loss: 0.0183, rpn_loss_cls: 0.0018, rpn_loss_box: 0.0
166, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.000100
speed: 0.570s / iter
iter: 69980 / 70000, total loss: 0.0026, rpn_loss_cls: 0.0016, rpn_loss_box: 0.0
010, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.000100
speed: 0.570s / iter
iter: 69990 / 70000, total loss: 0.0800, rpn_loss_cls: 0.0228, rpn_loss_box: 0.0
572, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.000100
speed: 0.570s / iter
iter: 70000 / 70000, total loss: 0.0112, rpn_loss_cls: 0.0073, rpn_loss_box: 0.0
039, loss_cls: 0.0000, loss_box: 0.0000, lr: 0.000100
speed: 0.570s / iter
Wrote snapshot to: /media/data/liuzy/Faster-RCNN_TF/output/faster_rcnn_end2end/v
oc_2007_trainval/VGGnet_fast_rcnn_iter_70000.ckpt
done solving
user2@ubuntu: /media/data/liuzy/Faster-RCNN_TF$

```

图 8.7 Faster R-CNN 图像目标检测案例程序在训练结束时的信息

Faster R-CNN 网络的测试命令如图 8.8 所示，测试结果如图 8.9 所示。从图 8.9 中可以看出，在 VOC 2007 的测试集上，Faster R-CNN 的总平均准确率为 68.11%，其中飞机类别的平均准确率为 69.80%，自行车类别的平均准确率为 78.83%，鸟类别的平均准确率为 65.67%。

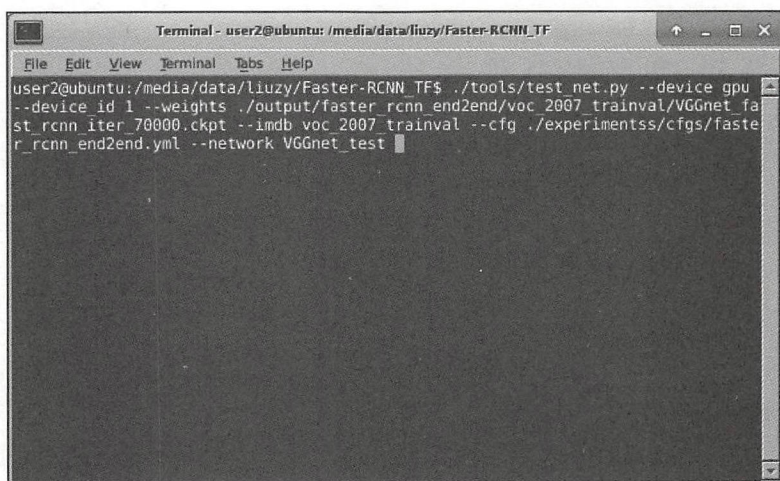


图 8.8 Faster R-CNN 图像目标检测案例程序的测试命令

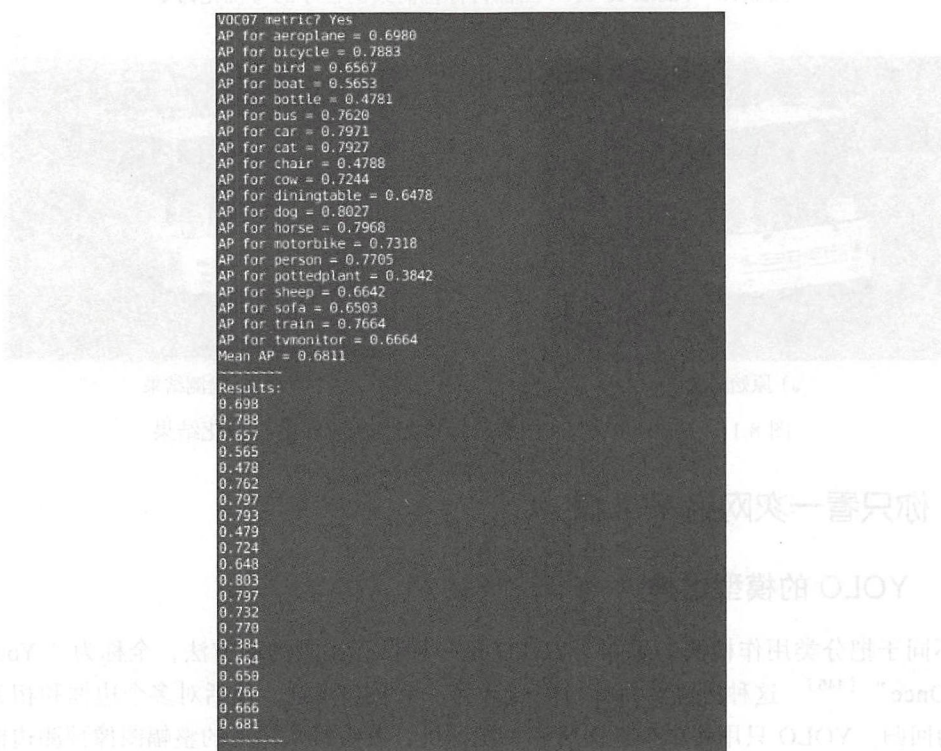


图 8.9 Faster R-CNN 图像目标检测案例程序的测试结果

如果要进行可视化，参照图 8.10 执行有关命令，图 8.11 给出了一幅原始图像和相应的可视化结果。

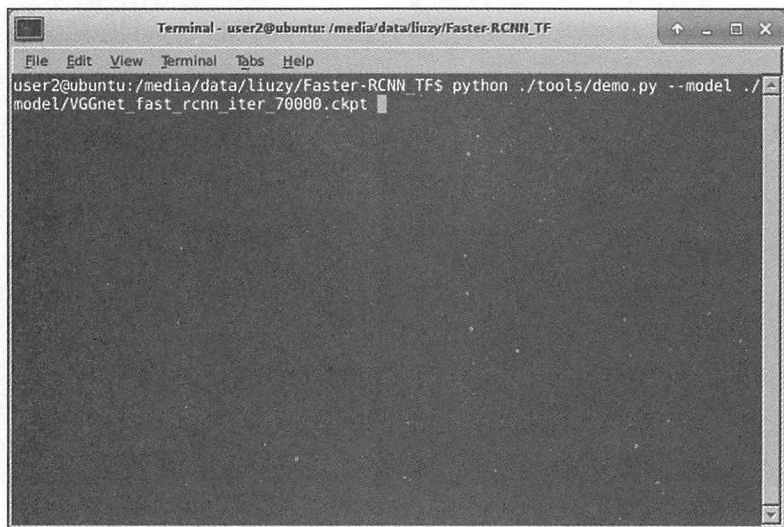


图 8.10 Faster R-CNN 图像目标检测案例程序的可视化命令



a) 原始图像



b) 检测结果

图 8.11 Faster R-CNN 图像目标检测案例程序的可视化结果

8.4 你只看一次网络 YOLO

8.4.1 YOLO 的模型结构

不同于把分类用作检测的思路，YOLO 是一种目标检测的新方法，全称为 “You Only Look Once”^[145]。这种方法将目标检测设计为一个回归问题，包括对多个边框和相关类别概率的回归。YOLO 只用一个神经网络和一次评价，就直接从输入的整幅图像预测边框和类别概率。正因为整个检测流程是一个网络，所以它可以直接进行端对端的优化。作为一种统一结构，YOLO 的运行速度非常快。基准 YOLO 模型每秒可以实时地处理 45 帧图像。一个较小版本——Fast YOLO，处理速度可达每秒 155 帧图像，同时保持较高的目标检测平均准确率（mAP），而且应用于训练领域外的图像时泛化能力也较强。

YOLO 进行目标检测的基本思路如图 8.12 所示,说明检测过程的一个具体例子如图 8.13 所示。YOLO 检测系统先将输入图像分成 $S \times S$ 个网格。每个网格负责检测中心落在其中的对象目标,并预测 B 个边框及相应的置信得分。置信得分表示一个边框含有对象目标的可信程度和精确程度有多大,并形式地定义为 $\Pr(\text{Object}) * \text{IoU}_{\text{pred}}^{\text{truth}}$ 。如果不包含目标,则得分应该为 0;否则,得分为预测边框与真实边框的 IoU。每个边框都有 5 个预测值: x 、 y 、 w 、 h 、confidence。其中, (x, y) 表示边框中心相对网格边界的位置坐标,实际上是用 0 到 1 之间的比例系数来表示的; w 和 h 是相对整幅图像预测的宽度和高度,实际上也是用 0 到 1 之间的比例系数来表示的; confidence 是置信度,定义为预测边框与真实边框的 IoU 值。此外,每个网格还要预测 C 个条件类别概率 $\Pr(\text{Class}|\text{Object})$,表示网格包含对象目标的类别概率。每个网格只预测一组类别概率,与预测的边框个数及大小无关。因此,YOLO 的预测结果(或网络输出)可以编码为一个 $S \times S \times (S \times 5 + C)$ 的张量。比如,在 PASCAL VOC 数据集上,可取 $S = 7$ 、 $B = 2$ 、 $C = 20$,表示将图像划分为 $S \times S = 7 \times 7 = 49$ 个网格,每个网格预测 $B = 2$ 个边框 $(x, y, w, h, \text{confidence})$ 和 $C = 20$ 个类别概率,YOLO 的最终预测结果是一个 $7 \times 7 \times 30$ 的张量。

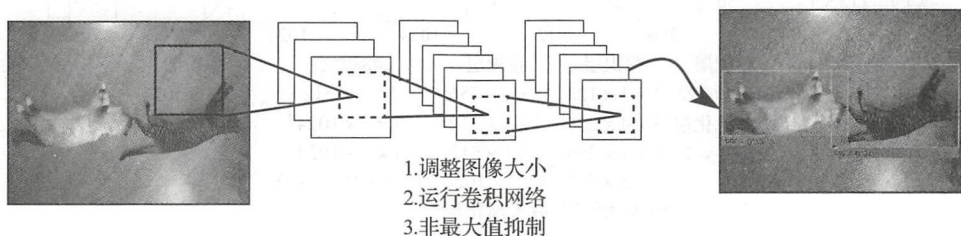


图 8.12 YOLO 的目标检测思路

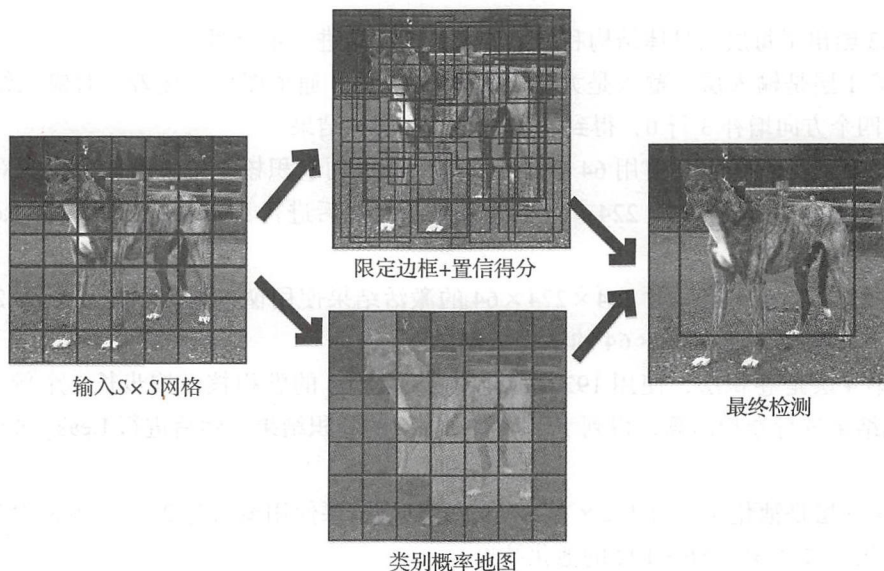


图 8.13 YOLO 的目标检测过程示例

在测试阶段，还要给每个边框计算类别有关的置信得分，用来编码类别在边框出现的概率和预测边框对对象目标的拟合好坏。具体计算方法是将条件类别概率与边框的置信度相乘，即

$$\Pr(\text{Class}_i | \text{Object}) * \Pr(\text{Object}) * \text{IoU}_{\text{pred}}^{\text{truth}} = \Pr(\text{Class}_i) * \text{IoU}_{\text{pred}}^{\text{truth}} \quad (8.8)$$

YOLO 的网络结构是在 GoogLeNet 的基础上建立的，如图 8.14 所示。这个网络用来处理 PASCAL VOC 数据集，共有 31 层（包含输入层），其中有 24 个卷积层、4 个池化层和 2 个全连接层。

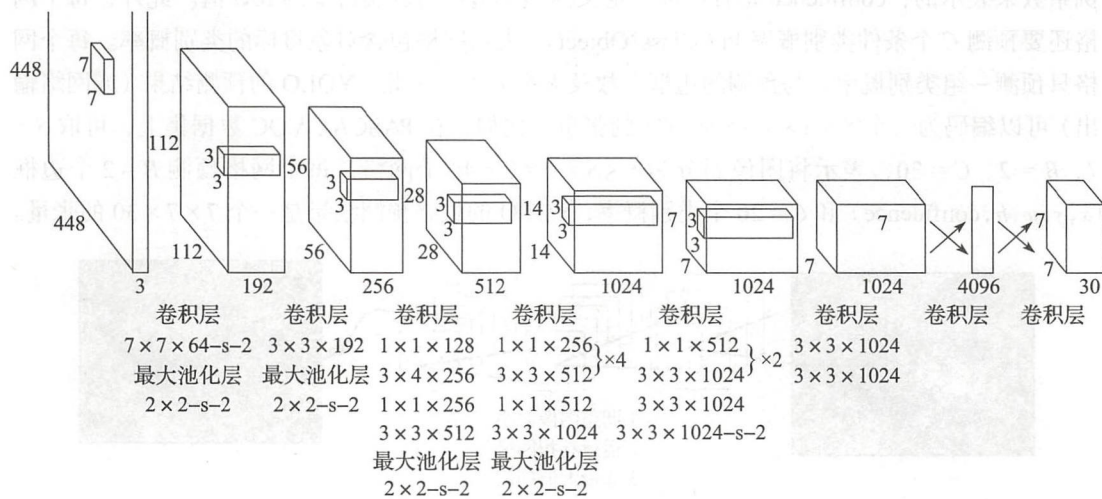


图 8.14 YOLO 的网络结构

表 8.3 给出了每层的具体结构和详细参数，下面是进一步说明。

1) 第 1 层是输入层，输入是大小为 448×448 的 3 通道图像。接着，对输入结果进行上下左右四个方向增补 3 行 0，得到 $454 \times 454 \times 3$ 增补结果。

2) 第 2 层是卷积层，使用 64 个大小为 $7 \times 7 \times 3$ 的卷积核，按步长 2 个像素对增补结果进行卷积运算，得到 $224 \times 224 \times 64$ 的卷积结果，然后进行 Leaky ReLU（渗漏 ReLU）激活函数运算。

3) 第 3 层是池化层，对 $224 \times 224 \times 64$ 的激活结果使用窗口为 2×2 、步长为 2 个像素的最大池化，得到 $112 \times 112 \times 64$ 的池化结果。

4) 第 4 层是卷积层，使用 192 个大小为 $3 \times 3 \times 64$ 的卷积核，按步长 1 个像素对第 3 层的池化结果进行卷积运算，得到 $112 \times 112 \times 192$ 的卷积结果，然后进行 Leaky ReLU 激活函数运算。

5) 第 5 层是池化层，对 $112 \times 112 \times 192$ 的激活结果使用窗口为 2×2 、步长为 2 个像素的最大池化，得到 $56 \times 56 \times 192$ 的池化结果。

6) 第 6 层是卷积层，使用 128 个大小为 $1 \times 1 \times 192$ 的卷积核，按步长 1 个像素对第 5

层的池化结果进行卷积运算,得到 $56 \times 56 \times 128$ 的卷积结果,然后进行 Leaky ReLU 激活函数运算。

7) 第7层是卷积层,使用256个大小为 $3 \times 3 \times 128$ 的卷积核,按步长1个像素对第6层的激活结果进行卷积运算,得到 $56 \times 56 \times 256$ 的卷积结果,然后进行 Leaky ReLU 激活函数运算。

8) 第8层是卷积层,使用256个大小为 $1 \times 1 \times 256$ 的卷积核,按步长1个像素对第7层的激活结果进行卷积运算,得到 $56 \times 56 \times 256$ 的卷积结果,然后进行 Leaky ReLU 激活函数运算。

9) 第9层是卷积层,使用512个大小为 $3 \times 3 \times 256$ 的卷积核,按步长1个像素对第8层的激活结果进行卷积运算,得到 $56 \times 56 \times 512$ 的卷积结果,然后进行 Leaky ReLU 激活函数运算。

10) 第10层是池化层,对 $56 \times 56 \times 512$ 的激活结果使用窗口为 2×2 、步长为2个像素的最大池化,得到 $28 \times 28 \times 512$ 的池化结果。

11) 第11层是卷积层,使用256个大小为 $1 \times 1 \times 512$ 的卷积核,按步长1个像素对第10层的池化结果进行卷积运算,得到 $28 \times 28 \times 256$ 的卷积结果,然后进行 Leaky ReLU 激活函数运算。

12) 第12层是卷积层,使用512个大小为 $3 \times 3 \times 256$ 的卷积核,按步长1个像素对第11层的激活结果进行卷积运算,得到 $28 \times 28 \times 512$ 的卷积结果,然后进行 Leaky ReLU 激活函数运算。

13) 第13层是卷积层,使用256个大小为 $1 \times 1 \times 512$ 的卷积核,按步长1个像素对第12层的激活结果进行卷积运算,得到 $28 \times 28 \times 256$ 的卷积结果,然后进行 Leaky ReLU 激活函数运算。

14) 第14层是卷积层,使用512个大小为 $3 \times 3 \times 256$ 的卷积核,按步长1个像素对第13层的激活结果进行卷积运算,得到 $28 \times 28 \times 512$ 的卷积结果,然后进行 Leaky ReLU 激活函数运算。

15) 第15层是卷积层,使用256个大小为 $1 \times 1 \times 512$ 的卷积核,按步长1个像素对第14层的激活结果进行卷积运算,得到 $28 \times 28 \times 256$ 的卷积结果,然后进行 Leaky ReLU 激活函数运算。

16) 第16层是卷积层,使用512个大小为 $3 \times 3 \times 256$ 的卷积核,按步长1个像素对第15层的激活结果进行卷积运算,得到 $28 \times 28 \times 512$ 的卷积结果,然后进行 Leaky ReLU 激活函数运算。

17) 第17层是卷积层,使用256个大小为 $1 \times 1 \times 512$ 的卷积核,按步长1个像素对第16层的激活结果进行卷积运算,得到 $28 \times 28 \times 256$ 的卷积结果,然后进行 Leaky ReLU 激活函数运算。

18) 第18层是卷积层,使用512个大小为 $3 \times 3 \times 256$ 的卷积核,按步长1个像素对第

17层的激活结果进行卷积运算，得到 $28 \times 28 \times 512$ 的卷积结果，然后进行 Leaky ReLU 激活函数运算。

19) 第19层是卷积层，使用256个大小为 $1 \times 1 \times 512$ 的卷积核，按步长1个像素对第18层的激活结果进行卷积运算，得到 $28 \times 28 \times 256$ 的卷积结果，然后进行 Leaky ReLU 激活函数运算。

20) 第20层是卷积层，使用1024个大小为 $3 \times 3 \times 256$ 的卷积核，按步长1个像素对第19层的激活结果进行卷积运算，得到 $28 \times 28 \times 1024$ 的卷积结果，然后进行 Leaky ReLU 激活函数运算。

21) 第21层是池化层，对 $28 \times 28 \times 1024$ 的激活结果使用窗口为 2×2 、步长为2个像素的最大池化，得到 $108 \times 108 \times 1024$ 的池化结果。

22) 第22层是卷积层，使用512个大小为 $1 \times 1 \times 1024$ 的卷积核，按步长1个像素对第21层的池化结果进行卷积运算，得到 $14 \times 14 \times 512$ 的卷积结果，然后进行 Leaky ReLU 激活函数运算。

23) 第23层是卷积层，使用1024个大小为 $3 \times 3 \times 512$ 的卷积核，按步长1个像素对第22层的激活结果进行卷积运算，得到 $14 \times 14 \times 1024$ 的卷积结果，然后进行 Leaky ReLU 激活函数运算。

24) 第24层是卷积层，使用512个大小为 $1 \times 1 \times 1024$ 的卷积核，按步长1个像素对第23层的激活结果进行卷积运算，得到 $14 \times 14 \times 512$ 的卷积结果，然后进行 Leaky ReLU 激活函数运算。

25) 第25层是卷积层，使用1024个大小为 $3 \times 3 \times 512$ 的卷积核，按步长1个像素对第24层的激活结果进行卷积运算，得到 $14 \times 14 \times 1024$ 的卷积结果，然后进行 Leaky ReLU 激活函数运算。

26) 第26层是卷积层，使用1024个大小为 $3 \times 3 \times 1024$ 的卷积核，按步长1个像素对第25层的激活结果进行卷积运算，得到 $14 \times 14 \times 1024$ 的卷积结果，然后进行 Leaky ReLU 激活函数运算，接着再对激活结果进行上下左右四个方向增补一行0，得到 $16 \times 16 \times 1024$ 增补结果。

27) 第27层是卷积层，使用1024个大小为 $3 \times 3 \times 1024$ 的卷积核，按步长2个像素对第26层的增补结果进行卷积运算，得到 $7 \times 7 \times 1024$ 的卷积结果，然后进行 Leaky ReLU 激活函数运算。

28) 第28层是卷积层，使用1024个大小为 $3 \times 3 \times 1024$ 的卷积核，按步长1个像素对第27层的激活结果进行卷积运算，得到 $7 \times 7 \times 1024$ 的卷积结果。然后，对卷积结果使用 Leaky ReLU 激活函数，得到激活结果。

29) 第29层是卷积层，使用1024个大小为 $3 \times 3 \times 1024$ 的卷积核，按步长1个像素对第28层的激活结果进行卷积运算，得到 $7 \times 7 \times 1024$ 的卷积结果，然后进行 Leaky ReLU 激活函数运算，接着再对激活结果进行平铺展开，得到展开结果。

30) 第30层是全连接层, 使用512个神经元, 对第29层的展开结果进行全连接处理, 然后进行 Leaky ReLU 激活函数运算。

31) 第31层是全连接层, 使用4096个 Leaky ReLU 神经元对第30层的展开结果进行全连接处理, 并以0.5的概率产生丢失输出(dropout)的结果。

32) 最后一层是输出层, 利用第31层的丢失输出结果进行预测, 输出一个 $7 \times 7 \times 30$ 的张量。

注意, 在上述各层中, 最后一层使用线性激活函数, 其余层使用的是下面的 Leaky ReLU 激活函数:

$$f(x) = \begin{cases} x, & x > 0 \\ 0.1x, & x \leq 0 \end{cases} \quad (8.9)$$

表 8.3 YOLO 网络每层的详细结构和参数

层	图像大小 / num_filters/ 节点	卷积核大小	步长
输入	448 × 448		
卷积	64	7 × 7	2
最大池化		2 × 2	2
卷积	192	3 × 3	1
最大池化		2 × 2	2
卷积	128	1 × 1	1
卷积	256	3 × 3	1
卷积	256	1 × 1	1
卷积	512	3 × 3	1
最大池化		2 × 2	2
卷积	256	1 × 1	1
卷积	512	3 × 3	1
卷积	256	1 × 1	1
卷积	512	3 × 3	1
卷积	256	1 × 1	1
卷积	512	3 × 3	1
卷积	256	1 × 1	1
卷积	512	3 × 3	1
卷积	256	1 × 1	1
卷积	1024	3 × 3	1
最大池化		2 × 2	2
卷积	512	1 × 1	1
卷积	1024	3 × 3	1
卷积	512	1 × 1	1
卷积	1024	3 × 3	1
卷积	1024	3 × 3	1

(续)

层	图像大小 / num_filters/ 节点	卷积核大小	步长
卷积	1024	3×3	2
卷积	1024	3×3	1
卷积	1024	3×3	1
全连接层	4096		
全连接层	$7 \times 7 \times 30$		

在训练过程中，YOLO 网络优化的目标函数是下面的多项平方误差损失：

$$\begin{aligned}
 & \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{\text{obj}} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\
 & + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{\text{obj}} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] + \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{\text{obj}} (C_i - \hat{C}_i)^2 \quad (8.10) \\
 & + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{\text{noobj}} (C_i - \hat{C}_i)^2 + \sum_{i=0}^{S^2} \mathbf{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
 \end{aligned}$$

其中， $\mathbf{1}_i^{\text{obj}}$ 表示对象目标是否出现在第 i 个网格中； $\mathbf{1}_{ij}^{\text{obj}}$ 表示第 i 个网格中第 j 个边框预测器是否负责有关预测。注意，损失函数只有在网格中存在对象目标时才惩罚分类错误，也只有在预测器负责相应真实边框时才惩罚边框的坐标错误。同时，在损失函数中，通过设定 $\lambda_{\text{coord}} = 5$ 、 $\lambda_{\text{noobj}} = 0.5$ 来增强网格包含对象目标的边框坐标损失，减少不包含对象目标的置信度损失。因此，不难发现，YOLO 网络的损失主要是定位错误产生的。

虽然 YOLO 的基础模型有 24 个卷积层，但处理速度在 Titan X GPU 上也可以达到 45fps，即每秒 45 帧图像。它的一个简化版本称为快速 YOLO，只包含 9 个卷积层和更少的卷积核，速度更快，超过 150 fps。

如果把 YOLO 原来的模型称为 YOLO v1，那么 YOLO 还有一个改进模型称为 YOLO v2^[146]。与 YOLO v1 相比，YOLO v2 的不同之处和创新特点主要包括：①使用了块归一化；②使用了高分辨率分类器；③使用了与锚点边框的卷积；④使用了维数聚类确定 5 种锚点边框的先验高宽比；⑤使用了无全连接层的新网络结构；⑥使用了直接位置预测；⑦使用了细粒度特征；⑧使用了多尺度训练。

最后，YOLO v2 还有一个版本称为 YOLO 9000^[146]，特指对 ImageNet 和 COCO 数据集中的 9000 多个类别进行目标实时检测的系统。

8.4.2 YOLO 的 TensorFlow 代码实现及说明

本节利用 TensorFlow 实现的 YOLO v1 网络，程序的下载地址为 https://github.com/hizhangp/yolo_tensorflow。该程序主要包括 6 个文件：pascal_voc.py、timer.py、config.py、yolo_net.py、train.py 和 test.py。其中，pascal_voc.py 定义了网络的数据输入情况，包括对

VOC 2007 数据的读取、归一化、分块等操作，为训练和测试做准备。timer.py 定义了网络训练过程中总时间和剩余时间的计算和显示。config.py 和 yolo_net.py 共同定义了网络的结构情况，包括网络训练和测试时结构。其中，config.py 定义了网络的输入大小以及学习率等情况，yolo_net.py 定义了网络的各层结构。train.py 定义了网络的训练情况，包括训练数据的读入和存储路径、训练参数的设置、训练过程的打印显示和训练模型的保存等内容。test.py 定义了网络的测试情况，包括测试数据的读入和存储路径、已训练模型的调用和测试过程的打印显示等内容。注意：pascal_voc.py、timer.py、config.py、yolo_net.py、train.py 和 test.py 这 6 个程序可能并不是存放在同一个目录下。

1. pascal_voc.py 的代码及说明

```
import os
import xml.etree.ElementTree as ET
import numpy as np
import cv2
import pickle
import copy
import yolo.config as cfg

class pascal_voc(object):    # 定义 pascal_voc 这个类
    def __init__(self, phase, rebuild=False):    # 利用 yolo.config 设置初始化参数
        self.devkit_path = os.path.join(cfg.PASCAL_PATH, 'VOCdevkit')
        self.data_path = os.path.join(self.devkit_path, 'VOC2007')
        self.cache_path = cfg.CACHE_PATH
        self.batch_size = cfg.BATCH_SIZE
        self.image_size = cfg.IMAGE_SIZE
        self.cell_size = cfg.CELL_SIZE
        self.classes = cfg.CLASSES
        self.class_to_ind = dict(zip(self.classes, range(len(self.classes))))
        self.flipped = cfg.FLIPPED
        self.phase = phase
        self.rebuild = rebuild
        self.cursor = 0
        self.epoch = 1
        self.gt_labels = None
        self.prepare()

    def get(self):    # 对图像和标签进行分块
        images = np.zeros((self.batch_size, self.image_size, self.image_size, 3))
        labels = np.zeros((self.batch_size, self.cell_size, self.cell_size, 25))
        count = 0
        while count < self.batch_size:
            imname = self.gt_labels[self.cursor]['imname']
            flipped = self.gt_labels[self.cursor]['flipped']
            images[count, :, :, :] = self.image_read(imname, flipped)
            labels[count, :, :, :] = self.gt_labels[self.cursor]['label']
            count += 1
            self.cursor += 1
        if self.cursor >= len(self.gt_labels):
            np.random.shuffle(self.gt_labels)
```

```

        self.cursor = 0
        self.epoch += 1
    return images, labels

def image_read(self, imname, flipped=False):
    # 读入图像并对图像进行统一大小裁剪和归一化
    image = cv2.imread(imname)
    image = cv2.resize(image, (self.image_size, self.image_size))
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB).astype(np.float32)
    image = (image / 255.0) * 2.0 - 1.0
    if flipped:
        image = image[:, ::-1, :]
    return image

def prepare(self):    # 定义图像的预处理，随机打乱
    gt_labels = self.load_labels()
    if self.flipped:
        print('Appending horizontally-flipped training examples ...')
        gt_labels_cp = copy.deepcopy(gt_labels)
        for idx in range(len(gt_labels_cp)):
            gt_labels_cp[idx]['flipped'] = True
            gt_labels_cp[idx]['label'] = gt_labels_cp[idx]['label'][:, ::-1, :]
            for i in range(self.cell_size):
                for j in range(self.cell_size):
                    if gt_labels_cp[idx]['label'][i, j, 0] == 1:
                        gt_labels_cp[idx]['label'][i, j, 1] = self.image_\
                            size - 1 - gt_labels_cp[idx]['label'][i, j, 1]
        gt_labels += gt_labels_cp
    np.random.shuffle(gt_labels)
    self.gt_labels = gt_labels
    return gt_labels

def load_labels(self):    # 加载图像的标签信息
    cache_file = os.path.join(self.cache_path, 'pascal_' + self.phase +\
        '_gt_labels.pkl')
    if os.path.isfile(cache_file) and not self.rebuild:
        print('Loading gt_labels from: ' + cache_file)
        with open(cache_file, 'rb') as f:
            gt_labels = pickle.load(f)
        return gt_labels
    print('Processing gt_labels from: ' + self.data_path)
    if not os.path.exists(self.cache_path):
        os.makedirs(self.cache_path)
    if self.phase == 'train':
        txtname = os.path.join(self.data_path, 'ImageSets', 'Main', 'trainval.\
            txt')
    else:
        txtname = os.path.join(self.data_path, 'ImageSets', 'Main', 'test.txt')
    with open(txtname, 'r') as f:
        self.image_index = [x.strip() for x in f.readlines()]
    gt_labels = []
    for index in self.image_index:
        label, num = self.load_pascal_annotation(index)
        if num == 0:
            continue

```



```

        imname = os.path.join(self.data_path, 'JPEGImages', index + '.jpg')
        gt_labels.append({'imname': imname, 'label': label, 'flipped': False})
    print('Saving gt_labels to: ' + cache_file)
    with open(cache_file, 'wb') as f:
        pickle.dump(gt_labels, f)
    return gt_labels

def load_pascal_annotation(self, index): # 加载 VOC 数据集图像和边框信息
    imname = os.path.join(self.data_path, 'JPEGImages', index + '.jpg')
    im = cv2.imread(imname) # 读取图片
    h_ratio = 1.0 * self.image_size / im.shape[0]
    w_ratio = 1.0 * self.image_size / im.shape[1]
    label = np.zeros((self.cell_size, self.cell_size, 25))
    filename = os.path.join(self.data_path, 'Annotations', index + '.xml')
    tree = ET.parse(filename) # 解析成 ElementTree 类的对象
    objs = tree.findall('object') # 找到 tree 中查找目标
    for obj in objs:
        bbox = obj.find('bndbox')
        # Make pixel indexes 0-based
        x1 = max(min((float(bbox.find('xmin').text) - 1) * w_ratio,
                    self.image_size - 1), 0)
        y1 = max(min((float(bbox.find('ymin').text) - 1) * h_ratio,
                    self.image_size - 1), 0)
        x2 = max(min((float(bbox.find('xmax').text) - 1) * w_ratio,
                    self.image_size - 1), 0)
        y2 = max(min((float(bbox.find('ymax').text) - 1) * h_ratio,
                    self.image_size - 1), 0)
        cls_ind = self.class_to_ind[obj.find('name').text.lower().strip()]
        boxes = [(x2 + x1) / 2.0, (y2 + y1) / 2.0, x2 - x1, y2 - y1]
        x_ind = int(boxes[0] * self.cell_size / self.image_size)
        y_ind = int(boxes[1] * self.cell_size / self.image_size)
        if label[y_ind, x_ind, 0] == 1:
            continue
        label[y_ind, x_ind, 0] = 1
        label[y_ind, x_ind, 1:5] = boxes
        label[y_ind, x_ind, 5 + cls_ind] = 1
    return label, len(objs)

```

2. timer.py 的代码及说明

```

import time, datetime

class Timer(object): # 定义各个简单的计时器
    def __init__(self): # 初始化相关事件参数
        self.init_time = time.time()
        self.total_time = 0.
        self.calls = 0
        self.start_time = 0.
        self.diff = 0.
        self.average_time = 0.
        self.remain_time = 0.

    def tic(self): # 使用 time.time 代替 time.clock 作为起始时间
        self.start_time = time.time()

    def toc(self, average=True): # 计算程序已经运行的平均时间

```

```

self.diff = time.time() - self.start_time
self.total_time += self.diff
self.calls += 1
self.average_time = self.total_time / self.calls
if average:
    return self.average_time
else:
    return self.diff
def remain(self, iters, max_iters):    # 计算程序运行结束时剩余的时间
    if iters == 0:
        self.remain_time = 0
    else:
        self.remain_time = (time.time() - self.init_time) * \
            (max_iters - iters) / iters
    return str(datetime.timedelta(seconds=int(self.remain_time)))

```

3. config.py 的代码及说明

```

import os
# 加载相关路径和数据集参数信息
DATA_PATH = 'data'
PASCAL_PATH = os.path.join(DATA_PATH, 'pascal_voc')
CACHE_PATH = os.path.join(PASCAL_PATH, 'cache')
OUTPUT_DIR = os.path.join(PASCAL_PATH, 'output')
WEIGHTS_DIR = os.path.join(PASCAL_PATH, 'weight')
WEIGHTS_FILE = os.path.join(DATA_PATH, 'weights', 'YOLO_small.ckpt')
CLASSES = ['aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus', 'car', 'cat',
            'chair', 'cow', 'diningtable', 'dog', 'horse',
            'motorbike', 'person', 'pottedplant', 'sheep', 'sofa', 'train', 'tvmonitor']

FLIPPED = True
# 加载模型参数
IMAGE_SIZE = 448
CELL_SIZE = 7
BOXES_PER_CELL = 2
ALPHA = 0.1
DISP_CONSOLE = False
OBJECT_SCALE = 1.0
NOOBJECT_SCALE = 1.0
CLASS_SCALE = 2.0
COORD_SCALE = 5.0
# 训练时所用参数
GPU = ''
LEARNING_RATE = 0.0001
DECAY_STEPS = 30000
DECAY_RATE = 0.1
STAIRCASE = True
BATCH_SIZE = 45
MAX_ITER = 15000
SUMMARY_ITER = 10
SAVE_ITER = 1000
# 测试时所用参数

```

```
THRESHOLD = 0.2
IOU_THRESHOLD = 0.5
```

4. yolo_net.py 的代码及说明

```
import numpy as np
import tensorflow as tf
import yolo.config as cfg
slim = tf.contrib.slim

class YOLONet(object):    # 定义YOLO 网络结构
    def __init__(self, is_training=True):    # 初始化网络相关参数
        self.classes = cfg.CLASSES
        self.num_class = len(self.classes)
        self.image_size = cfg.IMAGE_SIZE
        self.cell_size = cfg.CELL_SIZE
        self.bboxes_per_cell = cfg.BOXES_PER_CELL
        self.output_size = (self.cell_size * self.cell_size) * (self.num_class \
+ self.bboxes_per_cell * 5)
        self.scale = 1.0 * self.image_size / self.cell_size
        self.boundary1 = self.cell_size * self.cell_size * self.num_class
        self.boundary2 = self.boundary1 + self.cell_size * self.cell_size * \
self.bboxes_per_cell
        self.object_scale = cfg.OBJECT_SCALE
        self.noobject_scale = cfg.NOOBJECT_SCALE
        self.class_scale = cfg.CLASS_SCALE
        self.coord_scale = cfg.COORD_SCALE
        self.learning_rate = cfg.LEARNING_RATE
        self.batch_size = cfg.BATCH_SIZE
        self.alpha = cfg.ALPHA
        self.offset = np.transpose(np.reshape(np.array(
            [np.arange(self.cell_size)] * self.cell_size * self.bboxes_per_cell),
            (self.bboxes_per_cell, self.cell_size, self.cell_size)), (1, 2, 0))
        self.images = tf.placeholder(tf.float32, [None, self.image_size,
self.image_size, 3], name='images')
        self.logits = self.build_network(self.images, num_outputs=self.output \
_size, alpha=self.alpha, is_training=is_training)
        if is_training:
            self.labels = tf.placeholder(tf.float32, [None, self.cell_size,
self.cell_size, 5 + self.num_class])
            self.loss_layer(self.logits, self.labels)
            self.total_loss = tf.losses.get_total_loss()
            tf.summary.scalar('total_loss', self.total_loss)
        def build_network(self, images, num_outputs, alpha, keep_prob=0.5, is_training=
True, scope='yolo'):
            with tf.variable_scope(scope):    # 具体定义网络各层结构
                with slim.arg_scope([slim.conv2d, slim.fully_connected],
                    activation_fn=leaky_relu(alpha),
                    weights_initializer=tf.truncated_normal_initializer(0.0, 0.01),
                    weights_regularizer=slim.l2_regularizer(0.0005)):
                    net = tf.pad(images, np.array([[0, 0], [3, 3], [3, 3], [0, 0]]),
                        name='pad_1')
```



```

net = slim.conv2d(net, 64, 7, 2, padding='VALID', scope='conv_2')
net = slim.max_pool2d(net, 2, padding='SAME', scope='pool_3')
net = slim.conv2d(net, 192, 3, scope='conv_4')
net = slim.max_pool2d(net, 2, padding='SAME', scope='pool_5')
net = slim.conv2d(net, 128, 1, scope='conv_6')
net = slim.conv2d(net, 256, 3, scope='conv_7')
net = slim.conv2d(net, 256, 1, scope='conv_8')
net = slim.conv2d(net, 512, 3, scope='conv_9')
net = slim.max_pool2d(net, 2, padding='SAME', scope='pool_10')
net = slim.conv2d(net, 256, 1, scope='conv_11')
net = slim.conv2d(net, 512, 3, scope='conv_12')
net = slim.conv2d(net, 256, 1, scope='conv_13')
net = slim.conv2d(net, 512, 3, scope='conv_14')
net = slim.conv2d(net, 256, 1, scope='conv_15')
net = slim.conv2d(net, 512, 3, scope='conv_16')
net = slim.conv2d(net, 256, 1, scope='conv_17')
net = slim.conv2d(net, 512, 3, scope='conv_18')
net = slim.conv2d(net, 512, 1, scope='conv_19')
net = slim.conv2d(net, 1024, 3, scope='conv_20')
net = slim.max_pool2d(net, 2, padding='SAME', scope='pool_21')
net = slim.conv2d(net, 512, 1, scope='conv_22')
net = slim.conv2d(net, 1024, 3, scope='conv_23')
net = slim.conv2d(net, 512, 1, scope='conv_24')
net = slim.conv2d(net, 1024, 3, scope='conv_25')
net = slim.conv2d(net, 1024, 3, scope='conv_26')
net = tf.pad(net, np.array([[0, 0], [1, 1], [1, 1], [0, 0]]),
name='pad_27')
net = slim.conv2d(net, 1024, 3, 2, padding='VALID', scope='conv_28')
net = slim.conv2d(net, 1024, 3, scope='conv_29')
net = slim.conv2d(net, 1024, 3, scope='conv_30')
net = tf.transpose(net, [0, 3, 1, 2], name='trans_31')
net = slim.flatten(net, scope='flat_32')
net = slim.fully_connected(net, 512, scope='fc_33')
net = slim.fully_connected(net, 4096, scope='fc_34')
net = slim.dropout(net, keep_prob=keep_prob, is_training=is_training,
scope='dropout_35')
net = slim.fully_connected(net, num_outputs, activation_fn=None,
scope='fc_36')

return net

def calc_iou(self, boxes1, boxes2, scope='iou'):    # 计算边框的 IoU
    with tf.variable_scope(scope):
        boxes1 = tf.stack([boxes1[:, :, :, :, 0] - boxes1[:, :, :, :, 2] / 2.0,
            # 拼接边框
            boxes1[:, :, :, :, 1] - boxes1[:, :, :, :, 3] / 2.0,
            boxes1[:, :, :, :, 0] + boxes1[:, :, :, :, 2] / 2.0,
            boxes1[:, :, :, :, 1] + boxes1[:, :, :, :, 3] / 2.0])
        boxes1 = tf.transpose(boxes1, [1, 2, 3, 4, 0])    # 调换数据维度
        boxes2 = tf.stack([boxes2[:, :, :, :, 0] - boxes2[:, :, :, :, 2] / 2.0,
            boxes2[:, :, :, :, 1] - boxes2[:, :, :, :, 3] / 2.0,
            boxes2[:, :, :, :, 0] + boxes2[:, :, :, :, 2] / 2.0,
            boxes2[:, :, :, :, 1] + boxes2[:, :, :, :, 3] / 2.0])

```

```

boxes2 = tf.transpose(boxes2, [1, 2, 3, 4, 0])      # 调换数据维度
lu = tf.maximum(boxes1[:, :, :, :, :2], boxes2[:, :, :, :, :2])
# 计算左上起点
rd = tf.minimum(boxes1[:, :, :, :, 2:], boxes2[:, :, :, :, 2:])
# 计算右下终点
intersection = tf.maximum(0.0, rd - lu)
inter_square = intersection[:, :, :, :, 0] * intersection[:, :, :, :, 1]
# 通过求交求面积
# 计算 boxes1 和 boxes2 的平方
square1 = (boxes1[:, :, :, :, 2] - boxes1[:, :, :, :, 0]) * \
(boxes1[:, :, :, :, 3] - boxes1[:, :, :, :, 1])
square2 = (boxes2[:, :, :, :, 2] - boxes2[:, :, :, :, 0]) * \
(boxes2[:, :, :, :, 3] - boxes2[:, :, :, :, 1])
union_square = tf.maximum(square1 + square2 - inter_square, 1e-10)
return tf.clip_by_value(inter_square / union_square, 0.0, 1.0)
# 将目标值压缩至 0 ~ 1
def loss_layer(self, predicts, labels, scope='loss_layer'):    # 定义损失函数
    with tf.variable_scope(scope):
        predict_classes = tf.reshape(predicts[:, :self.boundary1],
            [self.batch_size, self.cell_size, self.cell_size, self.num_class])
        predict_scales = tf.reshape(predicts[:, self.boundary1:self.boundary2],
            [self.batch_size, self.cell_size, self.cell_size, self.bboxes_per_cell])
        predict_bboxes = tf.reshape(predicts[:, self.boundary2:],
            [self.batch_size, self.cell_size, self.cell_size, self.bboxes_per_cell, 4])
        response = tf.reshape(labels[:, :, :, 0], [self.batch_size,
            self.cell_size, self.cell_size, 1])
        boxes = tf.reshape(labels[:, :, :, 1:5], [self.batch_size,
            self.cell_size, self.cell_size, 1, 4])
        boxes = tf.tile(boxes, [1, 1, 1, self.bboxes_per_cell, 1])\
            / self.image_size      # 拼接边框
        classes = labels[:, :, :, 5:]
        offset = tf.constant(self.offset, dtype=tf.float32)
        offset = tf.reshape(offset, [1, self.cell_size, self.cell_size,
            self.bboxes_per_cell])
        offset = tf.tile(offset, [self.batch_size, 1, 1, 1])
        predict_bboxes_tran = tf.stack([(predict_bboxes[:, :, :, :, 0] + offset) / \
            self.cell_size, (predict_bboxes[:, :, :, :, 1] + tf.transpose(offset, (0, \
            2, 1, 3))) / self.cell_size,
            tf.square(predict_bboxes[:, :, :, :, 2]),
            tf.square(predict_bboxes[:, :, :, :, 3])])
        predict_bboxes_tran = tf.transpose(predict_bboxes_tran, [1, 2, 3, 4, 0])
        iou_predict_truth = self.calc_iou(predict_bboxes_tran, boxes) # 计算 IoU
        object_mask = tf.reduce_max(iou_predict_truth, 3, keep_dims=True)
        # 计算第 3 维最大值
        object_mask = tf.cast((iou_predict_truth >= object_mask), tf.float32) * \
            response
        noobject_mask = tf.ones_like(object_mask, dtype=tf.float32) - object_mask
        boxes_tran = tf.stack([boxes[:, :, :, :, 0] * self.cell_size - offset,
            # 拼接边框
            boxes[:, :, :, :, 1] * self.cell_size - tf.transpose(
                offset, (0, 2, 1, 3)),

```

```

        tf.sqrt(boxes[:, :, :, :, 2])),
        tf.sqrt(boxes[:, :, :, :, 3]))
boxes_tran = tf.transpose(boxes_tran, [1, 2, 3, 4, 0])
class_delta = response * (predict_classes - classes) # 开始计算分类损失
class_loss = tf.reduce_mean(tf.reduce_sum(tf.square(class_delta), axis=
[1, 2, 3]), name='class_loss') * self.class_scale
object_delta = object_mask * (predict_scales - iou_predict_truth)
# 开始计算目标损失
object_loss = tf.reduce_mean(tf.reduce_sum(tf.square(object_delta),
axis=[1, 2, 3]), name='object_loss') * self.object_scale
noobject_delta = noobject_mask * predict_scales # 开始计算noobject损失
noobject_loss = tf.reduce_mean(tf.reduce_sum(tf.square(noobject_delta),
axis=[1, 2, 3]), name='noobject_loss') * self.noobject_scale
coord_mask = tf.expand_dims(object_mask, 4) # 开始计算定位损失
boxes_delta = coord_mask * (predict_boxes - boxes_tran)
coord_loss = tf.reduce_mean(tf.reduce_sum(tf.square(boxes_delta), axis=
[1, 2, 3, 4]), name='coord_loss') * self.coord_scale
tf.losses.add_loss(class_loss) # 损失求和
tf.losses.add_loss(object_loss)
tf.losses.add_loss(noobject_loss)
tf.losses.add_loss(coord_loss)
tf.summary.scalar('class_loss', class_loss)
tf.summary.scalar('object_loss', object_loss)
tf.summary.scalar('noobject_loss', noobject_loss)
tf.summary.scalar('coord_loss', coord_loss)
tf.summary.histogram('boxes_delta_x', boxes_delta[:, :, :, :, 0])
tf.summary.histogram('boxes_delta_y', boxes_delta[:, :, :, :, 1])
tf.summary.histogram('boxes_delta_w', boxes_delta[:, :, :, :, 2])
tf.summary.histogram('boxes_delta_h', boxes_delta[:, :, :, :, 3])
tf.summary.histogram('iou', iou_predict_truth)

def leaky_relu(alpha): # 定义激活函数 leaky_relu
    def op(inputs):
        return tf.maximum(alpha * inputs, inputs, name='leaky_relu')
    return op

```

5. train.py 的代码及说明

```

import tensorflow as tf
import datetime
import os
import argparse
import yolo.config as cfg
from yolo.yolo_net import YOLONet
from utils.timer import Timer
from utils.pascal_voc import pascal_voc
class Solver(object): # 定义 Solver 类
    def __init__(self, net, data): # 初始化训练数据
        self.net = net
        self.data = data
        self.weights_file = cfg.WEIGHTS_FILE
        self.max_iter = cfg.MAX_ITER

```



```

self.initial_learning_rate = cfg.LEARNING_RATE
self.decay_steps = cfg.DECAY_STEPS
self.decay_rate = cfg.DECAY_RATE
self.staircase = cfg.STAIRCASE
self.summary_iter = cfg.SUMMARY_ITER
self.save_iter = cfg.SAVE_ITER
self.output_dir = os.path.join(
    cfg.OUTPUT_DIR, datetime.datetime.now().strftime('%Y_%m_%d_%H_%M'))
if not os.path.exists(self.output_dir):
    os.makedirs(self.output_dir)
self.save_cfg()
self.variable_to_restore = tf.global_variables()
self.restorer = tf.train.Saver(self.variable_to_restore, max_to_keep=None)
self.saver = tf.train.Saver(self.variable_to_restore, max_to_keep=None)
self.ckpt_file = os.path.join(self.output_dir, 'save.ckpt')
self.summary_op = tf.summary.merge_all()
self.writer = tf.summary.FileWriter(self.output_dir, flush_secs=60)
self.global_step = tf.get_variable('global_step', [],
    initializer=tf.constant_initializer(0), trainable=False)
self.learning_rate = tf.train.exponential_decay(
    self.initial_learning_rate, self.global_step, self.decay_steps,
    self.decay_rate, self.staircase, name='learning_rate')
self.optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=self.learning_rate).minimize(self.net.total_loss,
    global_step=self.global_step)
self.ema = tf.train.ExponentialMovingAverage(decay=0.9999)
self.averages_op = self.ema.apply(tf.trainable_variables())
with tf.control_dependencies([self.optimizer]):
    self.train_op = tf.group(self.averages_op)
gpu_options = tf.GPUOptions()
config = tf.ConfigProto(gpu_options=gpu_options)
self.sess = tf.Session(config=config)
self.sess.run(tf.global_variables_initializer())
if self.weights_file is not None:
    print('Restoring weights from: ' + self.weights_file)
    self.restorer.restore(self.sess, self.weights_file)
self.writer.add_graph(self.sess.graph)

def train(self):
    train_timer = Timer()      # 调用 Timer 函数, 开始计算起始时间
    load_timer = Timer()      # 调用 Timer 函数, 开始计算剩余时间
    for step in range(1, self.max_iter + 1):
        load_timer.tic()
        images, labels = self.data.get()
        load_timer.toc()
        feed_dict = {self.net.images: images, self.net.labels: labels}
        if step % self.summary_iter == 0:
            if step % (self.summary_iter * 10) == 0:
                train_timer.tic()
                summary_str, loss, _ = self.sess.run([self.summary_op,
                    self.net.total_loss, self.train_op], feed_dict=feed_dict)
                train_timer.toc()

```

```

        log_str = ('{} Epoch: {}, Step: {}, Learning rate: {}, '
        ' Loss: {:.3f}\nSpeed: {:.3f}s/iter, '
        ' Load: {:.3f}s/iter, Remain: {}').format(
            datetime.datetime.now().strftime('%m/%d %H:%M:%S'),
            self.data.epoch,
            int(step), round(self.learning_rate.eval(session=
            self.sess), 6), loss,
            train_timer.average_time, load_timer.average_time,
            train_timer.remain(step, self.max_iter))
        print(log_str)
    else:
        train_timer.tic()
        summary_str, _ = self.sess.run( [self.summary_op, self.train_op],
        feed_dict=feed_dict)
        train_timer.toc()
        self.writer.add_summary(summary_str, step)
    else:
        train_timer.tic()
        self.sess.run(self.train_op, feed_dict=feed_dict)
        train_timer.toc()
    if step % self.save_iter == 0:
        print('{} Saving checkpoint file to: {}'.format(
            datetime.datetime.now().strftime('%m/%d %H:%M:%S'),
            self.output_dir))
        self.saver.save(self.sess, self.ckpt_file,
            global_step=self.global_step)

def save_cfg(self):
    # 定义保存模型文件
    with open(os.path.join(self.output_dir, 'config.txt'), 'w') as f:
        cfg_dict = cfg.__dict__
        for key in sorted(cfg_dict.keys()):
            if key[0].isupper():
                cfg_str = '{}: {}\n'.format(key, cfg_dict[key])
                f.write(cfg_str)

def update_config_paths(data_dir, weights_file):
    # 定义更新权值函数
    cfg.DATA_PATH = data_dir
    cfg.PASCAL_PATH = os.path.join(data_dir, 'pascal_voc')
    cfg.CACHE_PATH = os.path.join(cfg.PASCAL_PATH, 'cache')
    cfg.OUTPUT_DIR = os.path.join(cfg.PASCAL_PATH, 'output')
    cfg.WEIGHTS_DIR = os.path.join(cfg.PASCAL_PATH, 'weights')
    cfg.WEIGHTS_FILE = os.path.join(cfg.WEIGHTS_DIR, weights_file)

def main():
    # 定义训练函数的主函数
    parser = argparse.ArgumentParser()
    parser.add_argument('--weights', default="YOLO_small.ckpt", type=str)
    parser.add_argument('--data_dir', default="data", type=str)
    parser.add_argument('--threshold', default=0.2, type=float)
    parser.add_argument('--iou_threshold', default=0.5, type=float)
    parser.add_argument('--gpu', default='', type=str)
    args = parser.parse_args()
    if args.gpu is not None:
        cfg.GPU = args.gpu
    if args.data_dir != cfg.DATA_PATH:
        update_config_paths(args.data_dir, args.weights)

```

```

os.environ['CUDA_VISIBLE_DEVICES'] = cfg.GPU
yolo = YOLONet()
pascal = pascal_voc('train')
solver = Solver(yolo, pascal)
print('Start training ...')
solver.train()
print('Done training.')
if __name__ == '__main__':      # 启动主函数
    main()

```

6. test.py 的代码及说明

```

import tensorflow as tf
import numpy as np
import os
import cv2
import argparse
import yolo.config as cfg
from yolo.yolo_net import YOLONet
from utils.timer import Timer
class Detector(object):      # 定义 Detector 类
    def __init__(self, net, weight_file):      # 初始化 Detector 类的参数
        self.net = net
        self.weights_file = weight_file
        self.classes = cfg.CLASSES
        self.num_class = len(self.classes)
        self.image_size = cfg.IMAGE_SIZE
        self.cell_size = cfg.CELL_SIZE
        self.bboxes_per_cell = cfg.BOXES_PER_CELL
        self.threshold = cfg.THRESHOLD
        self.iou_threshold = cfg.IOU_THRESHOLD
        self.boundary1 = self.cell_size * self.cell_size * self.num_class
        self.boundary2 = self.boundary1 + self.cell_size * self.cell_size \
            * self.bboxes_per_cell
        self.sess = tf.Session()
        self.sess.run(tf.global_variables_initializer())
        self.saver = tf.train.Saver()
        self.saver.restore(self.sess, self.weights_file)
    def draw_result(self, img, result):      # 画出检测结果
        for i in range(len(result)):
            x = int(result[i][1])
            y = int(result[i][2])
            w = int(result[i][3] / 2)
            h = int(result[i][4] / 2)
            cv2.rectangle(img, (x - w, y - h), (x + w, y + h), (0, 255, 0), 2)
            cv2.rectangle(img, (x - w, y - h - 20), (x + w, y - h),
                (125, 125, 125), -1)
            cv2.putText(img, result[i][0] + ' : %.2f' % result[i][5],
                (x - w + 5, y - h - 7), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 0), 1)
    def detect(self, img):      # 定义检查函数
        img_h, img_w, _ = img.shape
        inputs = cv2.resize(img, (self.image_size, self.image_size))

```



```

inputs = cv2.cvtColor(inputs, cv2.COLOR_BGR2RGB).astype(np.float32)
inputs = (inputs / 255.0) * 2.0 - 1.0
inputs = np.reshape(inputs, (1, self.image_size, self.image_size, 3))
result = self.detect_from_cvmat(inputs)[0]
for i in range(len(result)):
    result[i][1] *= (1.0 * img_w / self.image_size)
    result[i][2] *= (1.0 * img_h / self.image_size)
    result[i][3] *= (1.0 * img_w / self.image_size)
    result[i][4] *= (1.0 * img_h / self.image_size)
return result
def detect_from_cvmat(self, inputs):      # 定义网络测试结果
net_output = self.sess.run(self.net.logits, feed_dict={self.net.images:\
inputs})
results = []
for i in range(net_output.shape[0]):
    results.append(self.interpret_output(net_output[i]))
return results
def interpret_output(self, output):      # 定义输出解释函数
probs = np.zeros((self.cell_size, self.cell_size, self.bboxes_per_cell,
self.num_class))
class_probs = np.reshape(output[0:self.boundary1], (self.cell_size,
self.cell_size, self.num_class))
scales = np.reshape(output[self.boundary1:self.boundary2],
(self.cell_size, self.cell_size, self.bboxes_per_cell))
boxes = np.reshape(output[self.boundary2:], (self.cell_size,
self.cell_size, self.bboxes_per_cell, 4))
offset = np.transpose(np.reshape(np.array(
[np.arange(self.cell_size)] * self.cell_size * self.bboxes_per_cell),
[self.bboxes_per_cell, self.cell_size, self.cell_size]), (1, 2, 0))
boxes[:, :, :, 0] += offset
boxes[:, :, :, 1] += np.transpose(offset, (1, 0, 2))
boxes[:, :, :, :2] = 1.0 * boxes[:, :, :, 0:2] / self.cell_size
boxes[:, :, :, 2:] = np.square(boxes[:, :, :, 2:])
boxes *= self.image_size
for i in range(self.bboxes_per_cell):
    for j in range(self.num_class):
        probs[:, :, i, j] = np.multiply(
            class_probs[:, :, j], scales[:, :, i])
filter_mat_probs = np.array(probs >= self.threshold, dtype='bool')
filter_mat_boxes = np.nonzero(filter_mat_probs)
boxes_filtered = boxes[filter_mat_boxes[0], filter_mat_boxes[1],
filter_mat_boxes[2]]
probs_filtered = probs[filter_mat_probs]
classes_num_filtered = np.argmax(filter_mat_probs, axis=3)[filter_mat_boxes\
[0], filter_mat_boxes[1], filter_mat_boxes[2]]
argsort = np.array(np.argsort(probs_filtered))[:, :-1]
boxes_filtered = boxes_filtered[argsort]
probs_filtered = probs_filtered[argsort]
classes_num_filtered = classes_num_filtered[argsort]
for i in range(len(boxes_filtered)):
    if probs_filtered[i] == 0:
        continue
    for j in range(i + 1, len(boxes_filtered)):

```

```

        if self.iou(bboxes_filtered[i], bboxes_filtered[j]) > self.iou_threshold:
            probs_filtered[j] = 0.0
        filter_iou = np.array(probs_filtered > 0.0, dtype='bool')
        bboxes_filtered = bboxes_filtered[filter_iou]
        probs_filtered = probs_filtered[filter_iou]
        classes_num_filtered = classes_num_filtered[filter_iou]
        result = []
        for i in range(len(bboxes_filtered)):
            result.append([self.classes[classes_num_filtered[i]], bboxes_filtered[i][0], bboxes_filtered[i][1], bboxes_filtered[i][2], bboxes_filtered[i][3], probs_filtered[i]])
        return result

def iou(self, box1, box2):
    # 定义 box1 和 box2 的 IoU
    tb = min(box1[0] + 0.5 * box1[2], box2[0] + 0.5 * box2[2]) - \
        max(box1[0] - 0.5 * box1[2], box2[0] - 0.5 * box2[2])
    lr = min(box1[1] + 0.5 * box1[3], box2[1] + 0.5 * box2[3]) - \
        max(box1[1] - 0.5 * box1[3], box2[1] - 0.5 * box2[3])
    if tb < 0 or lr < 0:
        intersection = 0
    else:
        intersection = tb * lr
    return intersection / (box1[2] * box1[3] + box2[2] * box2[3] - intersection)

def image_detector(self, imname, wait=0):
    # 定义 image_detector 函数
    detect_timer = Timer()
    image = cv2.imread(imname)
    detect_timer.tic()
    result = self.detect(image)
    detect_timer.toc()
    print('Average detecting time: {:.3f}s'.format(detect_timer.average_time))
    self.draw_result(image, result)
    cv2.imshow('Image', image)
    cv2.waitKey(wait)

def main():
    # 定义主函数
    parser = argparse.ArgumentParser()
    parser.add_argument('--weights', default="save.ckpt-15000", type=str)
    parser.add_argument('--weight_dir', default="pascal_voc/output/modell", type=str)
    parser.add_argument('--data_dir', default="data", type=str)
    parser.add_argument('--gpu', default='', type=str)
    args = parser.parse_args()
    os.environ['CUDA_VISIBLE_DEVICES'] = args.gpu
    yolo = YOLONet(False)
    weight_file = os.path.join(args.data_dir, args.weight_dir, args.weights)
    detector = Detector(yolo, weight_file)
    imname = 'test/person.jpg'
    detector.image_detector(imname)

if __name__ == '__main__':
    # 开始测试检测
    main()

```

8.4.3 YOLO 的图像目标检测案例及演示效果

本节描述一个利用 YOLO 在 TensorFlow 框架下进行目标检测的案例，其中用到的 VOC 2007 图像数据集可以根据表 1.2 提供的地址下载。

YOLO 先在 ImageNet 分类任务上进行预训练（以一半的图像尺寸 224×224 ），然后再将图像尺寸变为 448×448 ，用于检测任务。预训练的网络结构如图 8.14 所示，主要包括前 20 层卷积层、一个平均池化层，以及一个全连接层。在 ImageNet 2012 的验证数据集上的 top-5 准确度达到 88% 之后就把预训练的模型用于检测。在预训练的模型上，YOLO 又增加了 4 个卷积层以及 2 个全连接层，其中的参数是随机初始化的。另外，在输出时，根据图像的宽、高将边框的宽、高进行归一化，将值归一化到 $[0, 1]$ 的区间。同时，也将边框中的坐标 (x, y) 通过与网格宽、高的对应比例归一化到 $[0, 1]$ 的区间。

注意：需要特别强调一下 YOLO 代码的存放目录问题。主目录下有 4 个文件夹：yolo、utils、test 和 data，另加 2 个文件：train.py 和 test.py。文件夹 yolo 包含两个文件：config.py 和 yolo_net.py。文件夹 utils 也包含两个文件：pascal_voc.py 和 timer.py。文件夹 test 用来存放测试结果，比如图像 car.jpg。文件夹 data 则包含两个子文件夹：pascal_voc 和 weights。pascal_voc 进一步包含 3 个子文件夹：cache、output 和 VOCdevkit，其中 cache 存放训练图像对应标签的信息，output 存放训练过程的中间模型，VOCdevkit 存放 VOC 2007 数据集。weights 存放预训练好的参数文件 YOLO_small.ckpt，下载地址为 <https://drive.google.com/file/d/0B5aC8pI-akZUNVFZMmhmcVRpbTA/view?usp=sharing>。

YOLO 案例程序的核心就是 8.4.2 节的 6 个程序 pascal_voc.py、timer.py、config.py、yolo_net.py、train.py 和 test.py，其训练命令如图 8.15 所示。训练在迭代 15 000 次后结束，结果如图 8.16 所示，训练损失函数值为 0.0138，训练准确率为 1。测试命令如图 8.17 所示，测试例子如图 8.18 所示。需要注意的是，因为此版本的代码并未给出计算测试集的准确率过程，所以这里只给出了可视化的结果。

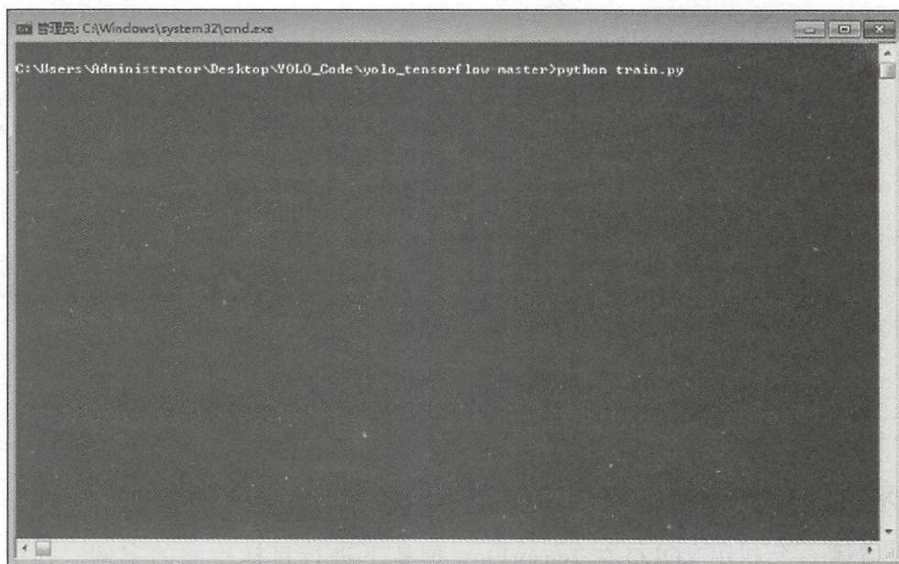


图 8.15 YOLO 图像目标检测案例程序的训练命令


```

管理员: C:\Windows\system32\cmd.exe
01/31 12:31:42 Epoch: 62. Step: 13700. Learning rate: 9.999999747378752e-05. Loss: 5.310
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 1:28:03
01/31 12:38:27 Epoch: 62. Step: 13800. Learning rate: 9.999999747378752e-05. Loss: 5.447
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 1:21:16
01/31 12:45:12 Epoch: 63. Step: 13900. Learning rate: 9.999999747378752e-05. Loss: 4.915
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 1:14:30
01/31 12:51:57 Epoch: 63. Step: 14000. Learning rate: 9.999999747378752e-05. Loss: 5.111
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 1:07:43
01/31 12:51:57 Saving checkpoint file to: data\pascal_voc\output\2018_01_30_21_03
01/31 12:58:49 Epoch: 64. Step: 14100. Learning rate: 9.999999747378752e-05. Loss: 4.671
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 1:00:57
01/31 13:05:34 Epoch: 64. Step: 14200. Learning rate: 9.999999747378752e-05. Loss: 4.565
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 0:54:11
01/31 13:12:19 Epoch: 65. Step: 14300. Learning rate: 9.999999747378752e-05. Loss: 5.260
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 0:47:24
01/31 13:19:04 Epoch: 65. Step: 14400. Learning rate: 9.999999747378752e-05. Loss: 5.065
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 0:40:38
01/31 13:25:49 Epoch: 66. Step: 14500. Learning rate: 9.999999747378752e-05. Loss: 4.620
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 0:33:51
01/31 13:32:33 Epoch: 66. Step: 14600. Learning rate: 9.999999747378752e-05. Loss: 4.922
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 0:27:05
01/31 13:39:19 Epoch: 67. Step: 14700. Learning rate: 9.999999747378752e-05. Loss: 4.867
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 0:20:19
01/31 13:46:04 Epoch: 67. Step: 14800. Learning rate: 9.999999747378752e-05. Loss: 5.173
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 0:13:32
01/31 13:52:49 Epoch: 67. Step: 14900. Learning rate: 9.999999747378752e-05. Loss: 5.061
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 0:06:46
01/31 13:59:34 Epoch: 68. Step: 15000. Learning rate: 9.999999747378752e-05. Loss: 4.027
Speed: 3.601s/iter, Load: 0.439s/iter, Remain: 0:00:00
01/31 13:59:34 Saving checkpoint file to: data\pascal_voc\output\2018_01_30_21_03
Done training.

C:\Users\Administrator\Desktop\YOLO_Code\yolo_tensorflow-master>

```

图 8.16 YOLO 图像目标检测案例程序的训练结果

```

管理员: C:\Windows\system32\cmd.exe

C:\Users\Administrator\Desktop\YOLO_Code\yolo_tensorflow-master>python test.py

```

图 8.17 YOLO 图像目标检测案例程序的测试命令

8.5 单次检测器 SSD

8.5.1 SSD 的模型结构

高性能目标检测系统的一般流程是：①对边框提出假设，②对每个边框重采像素或特征，③应用高质量分类器。这类系统大都使用了选择性搜索^[147]、Faster R-CNN^[142]和残差网络^[68]等技术，在 VOC、COCO 和 ILSVRC 等标准数据集上可以获得当时相对领先的结果。主要缺点是计算量太大，在嵌入式系统甚至在高端硬件上，都难以达到实时应用的检测速度。如果速度以每秒帧数（Frames Per Second, FPS）来衡量，最快的高精度检测器 Faster R-CNN 也只能达到每秒 7 帧。尝试建造速度更快的检测器，结果一般都是以牺牲检测精度为代价。

单次检测器（Single Shot Detector, SSD）是一种新型的深度神经网络目标检测器^[148]，不用对边框假设重采像素或特征，也不会损失精度，但速度比 Faster R-CNN 和 YOLO 都要快。SSD 在 VOC 2007 测试集上的检测速度为 59 FPS、平均准确率为 74.3%，而 Faster R-CNN 的速度只有 7FPS、平均准确率为 73.2%，YOLO 的速度为 45FPS、平均准确率为 63.4%。SSD 提高速度的根本改进措施是消除边框推荐和随后的像素或特征重采样阶段，还包括使用小卷积核在边框位置预测对象的类别和偏移，使用独立预测器（滤波器）负责不同高宽比的检测，并用这些滤波器在网络后期的多个特征图中执行多尺度检测。

SSD 的基础是一个前馈卷积网络，用来产生一个大小固定的边框集合，并对这些边框中的对象类别实例进行打分，再通过非极大值抑制步骤给出最后的检测结果。前面的网络层具有标准结构（可从任何分类层之前截断），用于高质量图像分类，称为基础网络。SSD 还给截断基础网络增加了用来检测的辅助结构，分别包括：

1) 多尺度特征图：在截断基础网络的末端添加卷积特征层，这些层的尺寸逐渐减小，但各层允许分别采用不同的卷积模型来进行多尺度检测的预测。注意，YOLO 只采用一个尺度的特征图^[145]。

2) 卷积预测器：每个添加的特征层（或从基础网络中选择的已有特征层）可以使用一组卷积核产生固定的预测集合。如图 8.19 所示，这些层标注在 SSD 网络结构的顶部。对于具有 p 个通道、大小为 $m \times n$ 的特征层，可以采用一个 $3 \times 3 \times p$ 的小卷积核来作为检测参数预测的基本元素，其作用是在每个位置上对类别打分，或者产生相对于默认边框坐标的形状偏移量。注意，YOLO 结构采用的是中间全连接层，而不是卷积层。

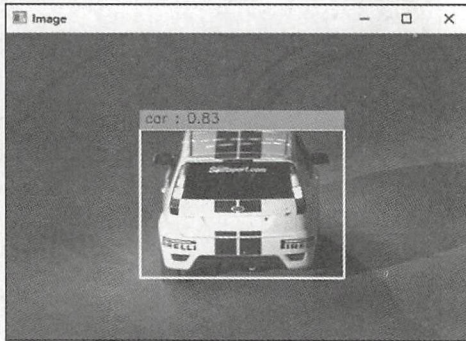


图 8.18 YOLO 图像目标检测案例程序的测试例子

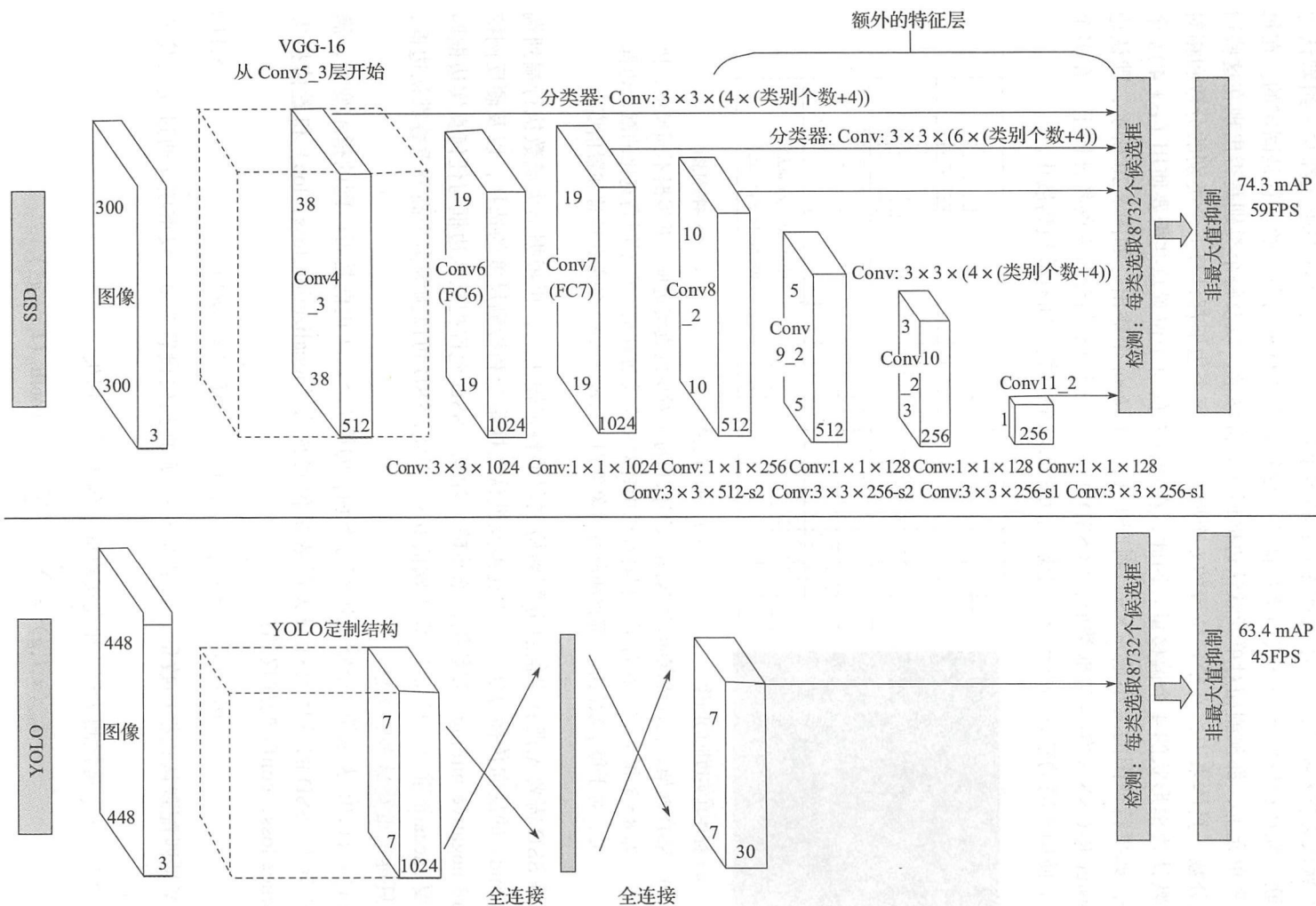


图 8.19 两种单次检测模型的比较: SSD 和 YOLO。SSD 模型在基础网络末端增加了几个特征层, 用来预测不同尺度的默认边框偏移、高度比和与它们关联的置信度。SSD 使用 300×300 的输入大小, 在 VOC 2007 的测试集上显著优于使用 448×448 输入大小的 YOLO, 而且速度更快

3) 默认边框和高宽比：对于网络顶部的多重特征图，每个特征图单元都与一组默认边框相关联。默认边框按卷积方式填满特征图，使得每个边框相对单元的位置是固定的。在每个特征图单元上，都预测相对单元默认边框形状的偏移量，以及每个类别是否出现在这些边框中的分数。具体来说，在给定位置的 k 个框中，对每个边框都计算 c 个类别的分数和相对于原始默认边框形状的 4 个偏移量。因此，在特征图的每个位置总共需要使用 $(c+4)k$ 个卷积核，如果特征图的大小是 $m \times n$ ，则产生 $(c+4)kmn$ 个输出。如图 8.20 所示，默认边框与 Faster R-CNN 的锚点边框类似，但 SSD 把它们应用到不同分辨率的特征图上。在多个特征图上使用不同的默认边框形状，把输出边框形状的可能空间有效地离散化。

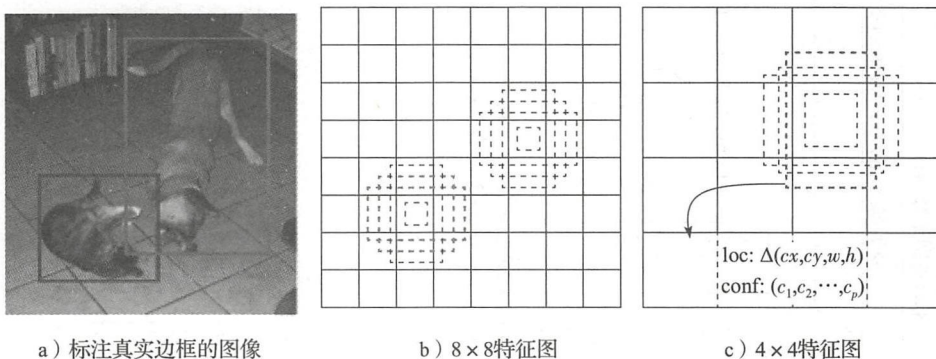


图 8.20 SSD 框架。在训练期间仅需要输入图像和每个对象的真实边框，并根据不同尺度，比如 8×8 和 4×4 ，在若干卷积特征图中针对每个位置评估一个不同高宽比的默认边框集。对于每个默认框，都预测所有对象类别 (c_1, c_2, \dots, c_p) 的形状偏移和置信度

训练 SSD 需要先把真实信息分配到检测器的特定输出上，再根据损失函数执行端到端 (end-to-end) 的反向传播算法，其中还涉及默认边框集合和检测尺度的选择，以及硬反例挖掘 (hard negative mining) 和数据扩增策略。注意：SSD 允许把默认边框与任何真实边框匹配，只要 Jaccard 重叠 (即 IoU) 大于阈值 0.5。因此，SSD 可以预测多个高分重叠默认边框，而不是只挑选重叠最多的边框。

用 $x_{ij}^p = \{1, 0\}$ 表示第 i 个默认边框与类别 p 的第 j 个真实边框是否匹配的指示变量，满足 $\sum_i x_{ij}^p \geq 1$ 。SSD 的总体目标损失函数是位置损失 (localization loss, loc) 和置信损失 (confidence loss, conf) 的加权和：

$$L(x, c, l, g) = \frac{1}{N} (L_{\text{conf}}(x, c) + \alpha L_{\text{loc}}(x, l, g)) \quad (8.11)$$

其中， N 是匹配默认边框的数量。如果 $N = 0$ ，则把损失设置为 0。权重项 α 通过交叉验证设置为 1。

位置损失 L_{loc} 是预测边框 l 和真实边框 g 的平滑 L_1 损失：

$$L_{\text{loc}}(x, l, g) = \sum_{i \in \text{Pos}} \sum_{m \in \{cx, cy, w, h\}} x_{ij}^k \text{smooth}_{L_1}(l_i^m - \hat{g}_j^m)$$

其中, smooth_{L_1} 的定义参见式 (8.1)。回归边框 \hat{g} 与真实边框 g 、默认边框 $d = (cx, cy, w, h)$ 之间的关系为

$$\begin{cases} \hat{g}_j^{cx} = (g_j^{cx} - d_i^{cx})/d_i^w, & \hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy})/d_i^h \\ \hat{g}_j^w = \log\left(\frac{g_j^w}{d_i^w}\right), & \hat{g}_j^h = \log\left(\frac{g_j^h}{d_i^h}\right) \end{cases} \quad (8.12)$$

置信损失 L_{conf} 是多类置信度的软最大损失。如果 Pos 和 Neg 分别表示所有正例边框和负例边框, 那么

$$L_{\text{conf}}(x, c) = - \sum_{i \in \text{Pos}} x_{ij}^p \log(\hat{c}_i^p) - \sum_{i \in \text{Neg}} \log(\hat{c}_i^0), \quad \hat{c}_i^p = \frac{\exp(c_i^p)}{\sum_p \exp(c_i^p)} \quad (8.13)$$

在 SSD 框架中, 默认边框虽然不必与每层的实际感受野相对应, 但是如何最优地选择其尺度和高宽比是一个难题。SSD 采用的是一种平铺方法。假设要使用 m 个特征图做预测, 每个特征图的默认边框尺度可以按下式计算:

$$s_k = s_{\min} + \frac{s_{\max} - s_{\min}}{m - 1}(k - 1), \quad k \in [1, m] \quad (8.14)$$

其中, s_{\min} 为 0.2, s_{\max} 为 0.9。这意味着最底层的尺度为 0.2, 最高层为 0.9, 并且中间所有层是等间隔的。

SSD 把默认边框的高宽比表示为 $a_r \in \left\{1, 2, 3, \frac{1}{2}, \frac{1}{3}\right\}$ 。因此, 每个默认边框的宽度和高度分别是

$$\begin{cases} w_k^a = s_k \sqrt{a_r} \\ h_k^a = s_k / \sqrt{a_r} \end{cases} \quad (8.15)$$

对于高宽比为 1 的情况, 还需要添加一个尺度为 $S'_k = \sqrt{s_k s_{k+1}}$ 的默认边框。因此, SSD 在每个特征图位置共产生 6 个默认边框。每个默认框的中心设置为 $\left(\frac{i + 0.5}{|f_k|}, \frac{j + 0.5}{|f_k|}\right)$, 其中, $|f_k|$ 是第 k 个正方形特征图的大小, $i, j \in [0, |f_k|)$ 。

SSD 的训练正例是通过匹配过程确立的。与真实边框满足匹配条件的默认边框都是正例, 其余的大多是反例。这在默认边框的数量较大时可能导致正反例数量的严重不平衡。解决办法是硬反例挖掘, 根据默认边框的置信度排序, 把反例和正例的比率控制在 3 : 1 以内。

为了提高关于目标大小和形状的鲁棒性, SSD 还采用了数据扩增的随机采样策略, 包括: ①使用整幅原始输入图像; ②采样图像块, 与目标的最小交并比为 0.1、0.3、0.5、0.7 或 0.9; ③随机采样图像块。每个采样块的大小为原始图像大小的 $[0.1, 1]$, 高宽比为 $1/2 \sim 2$ 。如果真实边框的中心在采样块内, 那么保留重叠部分。最后, 这些采样产生的图像块还要再经过光照扭曲^[149]、固定大小调整和按概率 0.5 水平翻转等操作处理。

8.5.2 SSD 的 TensorFlow 代码实现及说明

关于 SSD 的 TensorFlow 代码, 下载地址为 <https://github.com/balancap/SSD-Tensorflow>, 其

中包含 nets、datasets 和 preprocessing 等文件夹，以及 train_ssd_network.py、eval_ssd_network.py 和 tf_convert_data.py 等文件。考虑到代码间的相似性，下面仅详细说明训练网络文件 train_ssd_network.py 和文件夹 nets 中的模型定义文件 ssd_vgg_300.py。

1. 训练网络文件 train_ssd_network.py

```
import tensorflow as tf
from tensorflow.python.ops import control_flow_ops
from datasets import dataset_factory
from deployment import model_deploy
from nets import nets_factory
from preprocessing import preprocessing_factory
import tf_utils
slim = tf.contrib.slim
DATA_FORMAT = 'NCHW'
# ===== #
# SSD 网络标记
# ===== #
tf.app.flags.DEFINE_float('loss_alpha', 1., 'Alpha parameter in the loss function.')
tf.app.flags.DEFINE_float('negative_ratio', 3., 'Negative ratio in the loss function.')
tf.app.flags.DEFINE_float('match_threshold', 0.5, 'Matching threshold in the loss function.')
# ===== #
# 一般标记
# ===== #
tf.app.flags.DEFINE_string('train_dir', '/tmp/tfmodel/', 'Directory where checkpoints and event logs are written to.')
tf.app.flags.DEFINE_integer('num_clones', 1, 'Number of model clones to deploy.')
tf.app.flags.DEFINE_boolean('clone_on_cpu', False, 'Use CPUs to deploy clones.')
tf.app.flags.DEFINE_integer('num_readers', 4, 'The number of parallel readers that read data from the dataset.')
tf.app.flags.DEFINE_integer('num_preprocessing_threads', 4, 'The number of threads used to create the batches.')
tf.app.flags.DEFINE_integer('log_every_n_steps', 10, 'The frequency with which logs are print.')
tf.app.flags.DEFINE_integer('save_summaries_secs', 600, 'The frequency with which summaries are saved, in seconds.')
tf.app.flags.DEFINE_integer('save_interval_secs', 600, 'The frequency with which the model is saved, in seconds.')
tf.app.flags.DEFINE_float('gpu_memory_fraction', 0.8, 'GPU memory fraction to use.')
# ===== #
# 优化标记
# ===== #
tf.app.flags.DEFINE_float('weight_decay', 0.00004, 'The weight decay on the model weights.')
tf.app.flags.DEFINE_string('optimizer', 'rmsprop',
    'The name of the optimizer, one of "adadelata", "adagrad", "adam", "'ftrl", "momentum", "sgd" or "rmsprop".')
tf.app.flags.DEFINE_float('adadelata_rho', 0.95, 'The decay rate for adadelata.')
tf.app.flags.DEFINE_float('adagrad_initial_accumulator_value', 0.1, 'Starting value
```




```

for the AdaGrad accumulators.')
tf.app.flags.DEFINE_float('adam_beta1', 0.9, 'The exponential decay rate for the
1st moment estimates.')
tf.app.flags.DEFINE_float('adam_beta2', 0.999, 'The exponential decay rate for
the 2nd moment estimates.')
tf.app.flags.DEFINE_float('opt_epsilon', 1.0, 'Epsilon term for the optimizer.')
tf.app.flags.DEFINE_float('ftrl_learning_rate_power', -0.5, 'The learning rate
power.')
tf.app.flags.DEFINE_float('ftrl_initial_accumulator_value', 0.1, 'Starting value
for the FTRL accumulators.')
tf.app.flags.DEFINE_float('ftrl_l1', 0.0, 'The FTRL l1 regularization strength.')
tf.app.flags.DEFINE_float('ftrl_l2', 0.0, 'The FTRL l2 regularization strength.')
tf.app.flags.DEFINE_float('momentum', 0.9,
'The momentum for the MomentumOptimizer and RMSPropOptimizer.')
tf.app.flags.DEFINE_float('rmsprop_momentum', 0.9, 'Momentum.')
tf.app.flags.DEFINE_float('rmsprop_decay', 0.9, 'Decay term for RMSProp.')
#===== #
# 学习率标记
#===== #
tf.app.flags.DEFINE_string('learning_rate_decay_type', 'exponential',
'Specifies how the learning rate is decayed. One of "fixed",
"exponential", ' ' or "polynomial"')
tf.app.flags.DEFINE_float('learning_rate', 0.01, 'Initial learning rate.')
tf.app.flags.DEFINE_float('end_learning_rate', 0.0001,
'The minimal end learning rate used by a polynomial decay
learning rate.')
tf.app.flags.DEFINE_float('label_smoothing', 0.0, 'The amount of label smoothing.')
tf.app.flags.DEFINE_float('learning_rate_decay_factor', 0.94, 'Learning rate decay
factor.')
tf.app.flags.DEFINE_float('num_epochs_per_decay', 2.0, 'Number of epochs after
which learning rate decays.')
tf.app.flags.DEFINE_float('moving_average_decay', None, 'The decay to use for the
moving average.'
'If left as None, then moving averages are not used.')
#===== #
# 数据集标记
#===== #
tf.app.flags.DEFINE_string('dataset_name', 'imagenet', 'The name of the dataset
to load.')
tf.app.flags.DEFINE_integer('num_classes', 21, 'Number of classes to use in the
dataset.')
tf.app.flags.DEFINE_string('dataset_split_name', 'train', 'The name of the train/
test split.')
tf.app.flags.DEFINE_string('dataset_dir', None, 'The directory where the dataset
files are stored.')
tf.app.flags.DEFINE_integer('labels_offset', 0, 'An offset for the labels in the
dataset. This flag is primarily used to
'evaluate the VGG and ResNet architectures which do not use a background class
for the ImageNet dataset.')
tf.app.flags.DEFINE_string('model_name', 'ssd_300_vgg', 'The name of the architecture
to train.')

```



```

tf.app.flags.DEFINE_string('preprocessing_name', None, 'The name of the preprocessing
to use. If left as `None`, then the model_name flag is used.')
tf.app.flags.DEFINE_integer('batch_size', 32, 'The number of samples in each batch.')
tf.app.flags.DEFINE_integer('train_image_size', None, 'Train image size')
tf.app.flags.DEFINE_integer('max_number_of_steps', None, 'The maximum number of
training steps.')
#=====#
# 微调标记
#===== #
tf.app.flags.DEFINE_string('checkpoint_path', None, 'The path to a checkpoint from
which to fine-tune.')
tf.app.flags.DEFINE_string('checkpoint_model_scope', '../checkpoints',
                           'Model scope in the checkpoint. None if the same as the
                           trained model.')
tf.app.flags.DEFINE_string('checkpoint_exclude_scopes', None,
                           'Comma-separated list of scopes of variables to exclude when restoring
                           from a checkpoint.')
tf.app.flags.DEFINE_string('trainable_scopes', None,
                           'Comma-separated list of scopes to filter the set of variables to train. By default,
                           None would train all the variables.')
tf.app.flags.DEFINE_boolean('ignore_missing_vars', False,
                            'When restoring a checkpoint would ignore missing variables.')
FLAGS = tf.app.flags.FLAGS
#=====#
# 主要训练程序
#===== #
def main(_):
    if not FLAGS.dataset_dir:
        raise ValueError('You must supply the dataset directory with --dataset_dir')
    tf.logging.set_verbosity(tf.logging.DEBUG)
    with tf.Graph().as_default(): # 配置模型
        deploy_config = model_deploy.DeploymentConfig(num_clones=FLAGS.num_clones,
            clone_on_cpu=FLAGS.clone_on_cpu, replica_id=0, num_replicas=1,
            num_ps_tasks=0)
        with tf.device(deploy_config.variables_device()):
            global_step = slim.create_global_step()

        dataset = dataset_factory.get_dataset(
            FLAGS.dataset_name, FLAGS.dataset_split_name,
            FLAGS.dataset_dir) # 选择数据集
        dataset=tf.transpose(dataset,[0,2,3,1])

        # 获得 SSD 的网络结构及其锚
        ssd_class = nets_factory.get_network(FLAGS.model_name)
        ssd_params = ssd_class.default_params._replace(num_classes=\
            FLAGS.num_classes)
        ssd_net = ssd_class(ssd_params)
        ssd_shape = ssd_net.params.img_shape
        ssd_anchors = ssd_net.anchors(ssd_shape)

        preprocessing_name = FLAGS.preprocessing_name or FLAGS.model_name

```



```

# 选择预处理函数
image_preprocessing_fn = preprocessing_factory.get_preprocessing
(preprocessing_name, is_training=True)

tf_utils.print_configuration(FLAGS.__flags, ssd_params,
dataset.data_sources, FLAGS.train_dir)

#===== #
# 创建数据集提供程序和批处理
#===== #
with tf.device(deploy_config.inputs_device()):
    with tf.name_scope(FLAGS.dataset_name + '_data_provider'):
        provider = slim.dataset_data_provider.DatasetDataProvider
            (dataset, num_readers=FLAGS.num_readers, common_queue_capacity=\
                20 * FLAGS.batch_size,
                common_queue_min=10 * FLAGS.batch_size, shuffle=True)
        # 为 SSD 网络获得图像、标签和 bboxes
        [image, shape, glabels, gbboxes] = provider.get(['image', 'shape',
            'object/label', 'object/bbox'])
        # 对图像、标签和 bboxes 做预处理
        image, glabels, gbboxes = image_preprocessing_fn(image, glabels,
            gbboxes, out_shape=ssd_shape, data_format=DATA_FORMAT)
        # 对真实标签和 bboxes 进行编码
        gclasses, glocalisations, gscores = ssd_net.bboxes_encode(glabels,
            gbboxes, ssd_anchors)
        batch_shape = [1] + [len(ssd_anchors)] * 3

    r = tf.train.batch(tf_utils.reshape_list([image, gclasses, glocalisations,
        gscores]), # 训练迷你块和队列
        batch_size=FLAGS.batch_size, num_threads=FLAGS.num_preprocessing
            threads, capacity=5 * FLAGS.batch_size)
    b_image, b_gclasses, b_glocalisations, b_gscores = tf_utils.reshape_\
        list(r, batch_shape)

    # 中间队列: 所有在 GPU 上训练的唯一迷你块计算流程
    batch_queue = slim.prefetch_queue.prefetch_queue(
        tf_utils.reshape_list([b_image, b_gclasses, b_glocalisations,
            b_gscores]), capacity=2 * deploy_config.num_clones)

#===== #
# 定义运行在每个 GPU 上的模型
#===== #
def clone_fn(batch_queue): # 创建 network_fn 的多个副本, 使数据并行
    b_image, b_gclasses, b_glocalisations, b_gscores = \
        tf_utils.reshape_list(batch_queue.dequeue(), batch_shape)

# 创建 SSD 网络
arg_scope = ssd_net.arg_scope(weight_decay=FLAGS.weight_decay,
data_format=DATA_FORMAT)
with slim.arg_scope(arg_scope):
    predictions, localisations, logits, end_points = ssd_net.net
        (b_image, is_training=True)

```




```

        # 增加损失函数
        ssd_net.losses(logits, localisations, b_gclasses, b_glocalisations,
            b_gscores, match_threshold=FLAGS.match_threshold, negative_\
            ratio=FLAGS.negative_ratio, alpha=FLAGS.loss_alpha, label_smoothing=\
            FLAGS.label_smoothing)
        return end_points

# 收集初始信息
summaries = set(tf.get_collection(tf.GraphKeys.SUMMARIES))
# ===== #
# 从第一个副本增加信息
# ===== #
clones = model_deploy.create_clones(deploy_config, clone_fn, [batch_queue])
first_clone_scope = deploy_config.clone_scope(0)
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS, first_clone_scope)

# 为 end_points 增加信息
end_points = clones[0].outputs
for end_point in end_points:
    x = end_points[end_point]
    summaries.add(tf.summary.histogram('activations/' + end_point, x))
    summaries.add(tf.summary.scalar('sparsity/' + end_point, tf.nn.zero_\
        fraction(x)))

# 汇总为损失和额外损失
for loss in tf.get_collection(tf.GraphKeys.LOSSES, first_clone_scope):
    summaries.add(tf.summary.scalar(loss.op.name, loss))
for loss in tf.get_collection('EXTRA_LOSSES', first_clone_scope):
    summaries.add(tf.summary.scalar(loss.op.name, loss))

for variable in slim.get_model_variables(): # 汇总变量
    summaries.add(tf.summary.histogram(variable.op.name, variable))

# ===== #
# 配置移动平均变量
# ===== #
if FLAGS.moving_average_decay:
    moving_average_variables = slim.get_model_variables()
    variable_averages = tf.train.ExponentialMovingAverage(FLAGS.moving_\
        _average_decay, global_step)
else:
    moving_average_variables, variable_averages = None, None

# ===== #
# 设置优化过程
# ===== #
with tf.device(deploy_config.optimizer_device()):
    learning_rate = tf_utils.configure_learning_rate(FLAGS, dataset.num\
        _samples, global_step)
    optimizer = tf_utils.configure_optimizer(FLAGS, learning_rate)
    summaries.add(tf.summary.scalar('learning_rate', learning_rate))

```



```

if FLAGS.moving_average_decay:          # 使用训练器局部执行更新操作
    update_ops.append(variable_averages.apply(moving_average_variables))

    variables_to_train = tf_utils.get_variables_to_train(FLAGS)
    # 要训练的参数

    total_loss, clones_gradients = model_deploy.optimize_clones(clones,
                                                                optimizer, var_list=variables_to_train)
    # 计算全部损失 total_loss
    summaries.add(tf.summary.scalar('total_loss', total_loss))

    grad_updates = optimizer.apply_gradients(clones_gradients, global\
        _step=global_step) # 创建梯度更新
    update_ops.append(grad_updates)
    update_op = tf.group(*update_ops)
    train_tensor = control_flow_ops.with_dependencies([update_op], total\
        _loss, name='train_op')

    # 从第一个副本增加 summaries
    summaries |= set(tf.get_collection(tf.GraphKeys.SUMMARIES, first\
        _clone_scope))
    # 将所有的 summaries 合并到一起
    summary_op = tf.summary.merge(list(summaries), name='summary_op')

    # ===== #
    # 开始训练
    # ===== #
    gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=FLAGS.
        gpu_memory_fraction)
    config = tf.ConfigProto(log_device_placement=False, gpu_options=gpu\
        _options)
    saver = tf.train.Saver(max_to_keep=5, keep_checkpoint_every_n_hours=1.0,
        write_version=2, pad_step_number=False)
    slim.learning.train(train_tensor, logdir=FLAGS.train_dir, master='',
        is_chief=True, init_fn=tf_utils.get_init_fn(FLAGS), summary_op=summary\
        _op, number_of_steps=FLAGS.max_number_of_steps, log_every_n_steps=\
        FLAGS.log_every_n_steps, save_summaries_secs=FLAGS.save_summaries\
        secs, saver=saver, save_interval_secs=FLAGS.save_interval_secs, session\
        config=config, sync_optimizer=None)

if __name__ == '__main__':
    tf.app.run()

```

2. 模型定义文件 ssd_vgg_300.py

```

import math
from collections import namedtuple
import numpy as np
import tensorflow as tf
import tf_extended as tfe
from nets import custom_layers

```



```

from nets import ssd_common

slim = tf.contrib.slim

# ===== #
# SSD 类定义
# ===== #
SSDParams = namedtuple('SSDParameters', ['img_shape', 'num_classes', 'no_annotation_\
label', 'feat_layers', 'feat_shapes', 'anchor_size_bounds', 'anchor_sizes',\
'anchor_ratios', 'anchor_steps', 'anchor_offset', 'normalizations', 'prior_scaling'])
class SSDNet(object): # SSDNet 类, 实现基于 VGGNet 的 SSD 300 网络
    default_params = SSDParams(img_shape=(300, 300), num_classes=21, no_annotation_\
label=21,
        feat_layers=['block4', 'block7', 'block8', 'block9', 'block10', 'block11'],
        feat_shapes=[(38, 38), (19, 19), (10, 10), (5, 5), (3, 3), (1, 1)],
        anchor_size_bounds=[0.15, 0.90],
        # anchor_size_bounds=[0.20, 0.90],
        anchor_sizes=[(21., 45.), (45., 99.), (99., 153.), (153., 207.), (207., \
261.), (261., 315.)],
        anchor_ratios=[[2, .5], [2, .5, 3, 1./3], [2, .5, 3, 1./3], [2, .5, 3, \
1./3], [2, .5], [2, .5]],
        anchor_steps=[8, 16, 32, 64, 100, 300],
        anchor_offset=0.5,
        normalizations=[20, -1, -1, -1, -1, -1],
        prior_scaling=[0.1, 0.1, 0.2, 0.2])

    def __init__(self, params=None): # 初始化网络
        if isinstance(params, SSDParams):
            self.params = params
        else:
            self.params = SSDNet.default_params

    def net(self, inputs, is_training=True, update_feat_shapes=True, dropout_\
keep_prob=0.5, prediction_fn=slim.softmax, reuse=None, scope='ssd_300_vgg'):
        # SSD 网络定义
        r = ssd_net(inputs, num_classes=self.params.num_classes, feat_layers=\
self.params.feat_layers, anchor_sizes=self.params.anchor_sizes, anchor_\
ratios=self.params.anchor_ratios, normalizations=self.params.normalizations,\
is_training=is_training, dropout_keep_prob=dropout_keep_prob, prediction_\
fn=prediction_fn, reuse=reuse, scope=scope)
        # 更新特征形状 (至少进行尝试)
        if update_feat_shapes:
            shapes = ssd_feat_shapes_from_net(r[0], self.params.feat_shapes)
            self.params = self.params._replace(feat_shapes=shapes)
        return r

    def arg_scope(self, weight_decay=0.0005, data_format='NHWC'):
        return ssd_arg_scope(weight_decay, data_format=data_format)

    def arg_scope_caffe(self, caffe_scope): # 用于权值导入的 Caffe arg_scope
        return ssd_arg_scope_caffe(caffe_scope)

# ===== #
def update_feature_shapes(self, predictions): # 更新特征形状

```




```

    shapes = ssd_feat_shapes_from_net(predictions, self.params.feats_shapes)
    self.params = self.params._replace(feats_shapes=shapes)
def anchors(self, img_shape, dtype=np.float32):          # 计算默认的 anchor 框
    return ssd_anchors_all_layers(img_shape, self.params.feats_shapes,
                                   self.params.anchor_sizes, self.params.anchor_ratios, self.params.anchor_steps,
                                   self.params.anchor_offset, dtype)
def bboxes_encode(self, labels, bboxes, anchors, scope=None): # 编码标签和边框
    return ssd_common.tf_ssd_bboxes_encode(labels, bboxes, anchors,
                                             self.params.num_classes, self.params.no_annotation_label,
                                             ignore_threshold=0.5, prior_scaling=self.params.prior_scaling, scope=scope)
def bboxes_decode(self, feat_localizations, anchors, scope='ssd_bboxes_decode'):
    # 解码标签和边框
    return ssd_common.tf_ssd_bboxes_decode(feat_localizations, anchors,
                                             prior_scaling=self.params.prior_scaling, scope=scope)
def detected_bboxes(self, predictions, localisations, select_threshold=
None, nms_threshold=0.5, clipping_bbox=None, top_k=400, keep_top_k=200):
    # 输出 SSD 网络检测的边框
    # 从预测结果中选择 top_k 个 bboxes, 并对其裁剪
    rscores, rbboxes = ssd_common.tf_ssd_bboxes_select(predictions, localisations,
                                                         select_threshold=select_threshold,
                                                         num_classes=self.params.num_classes)
    rscores, rbboxes = tfe.bboxes_sort(rscores, rbboxes, top_k=top_k)
    rscores, rbboxes = tfe.bboxes_nms_batch(rscores, rbboxes, nms_threshold=
nms_threshold, keep_top_k=keep_top_k) # 使用非极大值抑制算法
    if clipping_bbox is not None:
        rbboxes = tfe.bboxes_clip(clipping_bbox, rbboxes)
    return rscores, rbboxes
def losses(self, logits, localisations, gclasses, glocalisations, gscores, match\
_threshold=0.5, negative_ratio=3., alpha=1., label_smoothing=0, scope='ssd_losses'):
    # 定义 SSD 网络损失
    return ssd_losses(logits, localisations, gclasses, glocalisations, gscores,
                       match_threshold=match_threshold, negative_ratio=negative\
_ratio, alpha=alpha, label_smoothing=label_smoothing, scope=scope)

# ===== #
# SSD 工具
# ===== #
def ssd_size_bounds_to_values(size_bounds, n_feat_layers, img_shape=(300, 300)):
    assert img_shape[0] == img_shape[1]
    img_size = img_shape[0]
    min_ratio = int(size_bounds[0] * 100)
    max_ratio = int(size_bounds[1] * 100)
    step = int(math.floor((max_ratio - min_ratio) / (n_feat_layers - 2)))
    sizes = [[img_size * size_bounds[0] / 2, img_size * size_bounds[0]]]
    # 从最小值开始
    for ratio in range(min_ratio, max_ratio + 1, step):
        sizes.append((img_size * ratio / 100., img_size * (ratio + step) / 100.))
    return sizes
def ssd_feat_shapes_from_net(predictions, default_shapes=None):
    # 从预测层获得特征形状
    feat_shapes = []

```



```

for l in predictions:
    if isinstance(l, np.ndarray): # 从 np 数组或者张量获得形状信息
        shape = l.shape
    else:
        shape = l.get_shape().as_list()
    shape = shape[1:4]
    if None in shape: # 问题：未决定形状
        return default_shapes
    else:
        feat_shapes.append(shape)
return feat_shapes

def ssd_anchor_one_layer(img_shape, feat_shape, sizes, ratios, step, offset=0.5,
dtype=np.float32):
    # 为特征层计算 SSD 默认的 anchor 框
    y, x = np.mgrid[0:feat_shape[0], 0:feat_shape[1]]
    y = (y.astype(dtype) + offset) * step / img_shape[0]
    x = (x.astype(dtype) + offset) * step / img_shape[1]

    y = np.expand_dims(y, axis=-1) # 扩展维度
    x = np.expand_dims(x, axis=-1)

    # 计算相对高和宽
    num_anchors = len(sizes) + len(ratios)
    h = np.zeros((num_anchors, ), dtype=dtype)
    w = np.zeros((num_anchors, ), dtype=dtype)
    # 添加比率等于 1 的第一批锚框
    h[0] = sizes[0] / img_shape[0]
    w[0] = sizes[0] / img_shape[1]
    di = 1
    if len(sizes) > 1:
        h[1] = math.sqrt(sizes[0] * sizes[1]) / img_shape[0]
        w[1] = math.sqrt(sizes[0] * sizes[1]) / img_shape[1]
        di += 1
    for i, r in enumerate(ratios):
        h[i+di] = sizes[0] / img_shape[0] / math.sqrt(r)
        w[i+di] = sizes[0] / img_shape[1] * math.sqrt(r)
    return y, x, h, w

def ssd_anchors_all_layers(img_shape, layers_shape, anchor_sizes, anchor_ratios,
                           anchor_steps, offset=0.5, dtype=np.float32):
    # 为所有特征层计算锚框
    layers_anchors = []
    for i, s in enumerate(layers_shape):
        anchor_bboxes = ssd_anchor_one_layer(img_shape, s, anchor_sizes[i], anchor\
        _ratios[i], anchor_steps[i], offset=offset, dtype=dtype)
        layers_anchors.append(anchor_bboxes)
    return layers_anchors

# ===== #
# 基于 VGGNet 的 SSD 300 的函数定义
# ===== #
def tensor_shape(x, rank=3): # 返回张量的维度

```



```

if x.get_shape().is_fully_defined():
    return x.get_shape().as_list()
else:
    static_shape = x.get_shape().with_rank(rank).as_list()
    dynamic_shape = tf.unstack(tf.shape(x), rank)
    return [s if s is not None else d
            for s, d in zip(static_shape, dynamic_shape)]

def ssd_multibox_layer(inputs, num_classes, sizes, ratios=[1], normalization=-1,
                        bn_normalization=False):
    # 创建 multibox 层, 返回类别和位置预测
    net = inputs
    if normalization > 0:
        net = custom_layers.l2_normalization(net, scaling=True)
    num_anchors = len(sizes) + len(ratios)      # 锚的数量

    # 位置
    num_loc_pred = num_anchors * 4
    loc_pred = slim.conv2d(net, num_loc_pred, [3, 3], activation_fn=None,
                           scope='conv_loc')
    loc_pred = custom_layers.channel_to_last(loc_pred)
    loc_pred = tf.reshape(loc_pred, tensor_shape(loc_pred, 4)[: -1] + [num_anchors, 4])
    # 类别预测
    num_cls_pred = num_anchors * num_classes
    cls_pred = slim.conv2d(net, num_cls_pred, [3, 3], activation_fn=None,
                           scope='conv_cls')
    cls_pred = custom_layers.channel_to_last(cls_pred)
    cls_pred = tf.reshape(cls_pred, tensor_shape(cls_pred, 4)[: -1] + [num_anchors,
                                num_classes])
    return cls_pred, loc_pred

def ssd_net(inputs, num_classes=SSDNet.default_params.num_classes,
            feat_layers=SSDNet.default_params.feat_layers,
            anchor_sizes=SSDNet.default_params.anchor_sizes,
            anchor_ratios=SSDNet.default_params.anchor_ratios,
            normalizations=SSDNet.default_params.normalizations,
            is_training=True, dropout_keep_prob=0.5, prediction_fn=slim.softmax,
            reuse=None, scope='ssd_300_vgg'):    # SSD 网络定义
    end_points = {}
    with tf.variable_scope(scope, 'ssd_300_vgg', [inputs], reuse=reuse):
        net = slim.repeat(inputs, 2, slim.conv2d, 64, [3, 3], scope='conv1')
        # 原始的 VGGNet-16 块
        end_points['block1'] = net
        net = slim.max_pool2d(net, [2, 2], scope='pool1')
        net = slim.repeat(net, 2, slim.conv2d, 128, [3, 3], scope='conv2')
        # 块 2
        end_points['block2'] = net
        net = slim.max_pool2d(net, [2, 2], scope='pool2')
        net = slim.repeat(net, 3, slim.conv2d, 256, [3, 3], scope='conv3') # 块 3
        end_points['block3'] = net
        net = slim.max_pool2d(net, [2, 2], scope='pool3')
        net = slim.repeat(net, 3, slim.conv2d, 512, [3, 3], scope='conv4') # 块 4
        end_points['block4'] = net

```



```

net = slim.max_pool2d(net, [2, 2], scope='pool4')
net = slim.repeat(net, 3, slim.conv2d, 512, [3, 3], scope='conv5') # 块5
end_points['block5'] = net
net = slim.max_pool2d(net, [3, 3], stride=1, scope='pool5')

# 另外的 SSD 块
net = slim.conv2d(net, 1024, [3, 3], rate=6, scope='conv6') # 块6
end_points['block6'] = net
net = tf.layers.dropout(net, rate=dropout_keep_prob, training=is_
training)
net = slim.conv2d(net, 1024, [1, 1], scope='conv7') # 块7: 1×1 卷积
end_points['block7'] = net
net = tf.layers.dropout(net, rate=dropout_keep_prob, training=is_training)
# 块8/9/10/11: 1×1 卷积和步长为2的3×3卷积(除了块10/11的3×3卷积)
end_point = 'block8'
with tf.variable_scope(end_point):
    net = slim.conv2d(net, 256, [1, 1], scope='conv1x1')
    net = custom_layers.pad2d(net, pad=(1, 1))
    net = slim.conv2d(net, 512, [3, 3], stride=2, scope='conv3x3', padding=
'VALID')
end_points[end_point] = net
end_point = 'block9'
with tf.variable_scope(end_point):
    net = slim.conv2d(net, 128, [1, 1], scope='conv1x1')
    net = custom_layers.pad2d(net, pad=(1, 1))
    net = slim.conv2d(net, 256, [3, 3], stride=2, scope='conv3x3', padding=
'VALID')
end_points[end_point] = net
end_point = 'block10'
with tf.variable_scope(end_point):
    net = slim.conv2d(net, 128, [1, 1], scope='conv1x1')
    net = slim.conv2d(net, 256, [3, 3], scope='conv3x3', padding='VALID')
end_points[end_point] = net
end_point = 'block11'
with tf.variable_scope(end_point):
    net = slim.conv2d(net, 128, [1, 1], scope='conv1x1')
    net = slim.conv2d(net, 256, [3, 3], scope='conv3x3', padding='VALID')
end_points[end_point] = net

# 预测和定位层
predictions = []
logits = []
localisations = []
for i, layer in enumerate(feats_layers):
    with tf.variable_scope(layer + '_box'):
        p, l = ssd_multibox_layer(end_points[layer], num_classes, anchor_
sizes[i], anchor_ratios[i], normalizations[i])
        predictions.append(prediction_fn(p))
        logits.append(p)
        localisations.append(l)

```

```

        return predictions, localisations, logits, end_points

ssd_net.default_image_size = 300

def ssd_arg_scope(weight_decay=0.0005, data_format='NHWC'): # 定义 VGGNet 参数
    with slim.arg_scope([slim.conv2d, slim.fully_connected], activation_fn=tf.nn.relu,
                        weights_regularizer=slim.l2_regularizer(weight_decay), weights_initializer=tf.contrib.layers.xavier_initializer(),
                        biases_initializer=tf.zeros_initializer()):
        with slim.arg_scope([slim.conv2d, slim.max_pool2d], padding='SAME', data_format=data_format):
            with slim.arg_scope([custom_layers.pad2d, custom_layers.l2_normalization,
                                custom_layers.channel_to_last], data_format=data_format) as sc:

                return sc

# ===== #
# Caffe 范围: 导入初始化的权值
# ===== #
def ssd_arg_scope_caffe(caffe_scope):
    # 默认网络参数范围
    with slim.arg_scope([slim.conv2d], activation_fn=tf.nn.relu,
                        weights_initializer=caffe_scope.conv_weights_init(),
                        biases_initializer=caffe_scope.conv_biases_init()):
        with slim.arg_scope([slim.fully_connected], activation_fn=tf.nn.relu):
            with slim.arg_scope([custom_layers.l2_normalization],
                                scale_initializer=caffe_scope.l2_norm_scale_init()):
                with slim.arg_scope([slim.conv2d, slim.max_pool2d], padding='SAME')
                    as sc:

                    return sc

# ===== #
# SSD 损失函数
# ===== #
def ssd_losses(logits, localisations, gclasses, glocalisations, gscores, match_threshold=0.5, negative_ratio=3.,
               alpha=1., label_smoothing=0., device='/cpu:0', scope=None):
    with tf.name_scope(scope, 'ssd_losses'):
        lshape = tfe.get_shape(logits[0], 5)
        num_classes = lshape[-1]
        batch_size = lshape[0]
        # 拉平所有向量
        flogits = []
        fgclasses = []
        fgcores = []
        flocalisations = []
        fglocalisations = []
        for i in range(len(logits)):
            flogits.append(tf.reshape(logits[i], [-1, num_classes]))
            fgclasses.append(tf.reshape(gclasses[i], [-1]))
            fgcores.append(tf.reshape(gscores[i], [-1]))
            flocalisations.append(tf.reshape(localisations[i], [-1, 4]))

```

```

        fglocalisations.append(tf.reshape(glocalisations[i], [-1, 4]))
# 拼接 carp
logits = tf.concat(flogits, axis=0)
gclasses = tf.concat(fgclasses, axis=0)
gscores = tf.concat(fgscores, axis=0)
localisations = tf.concat(flocalisations, axis=0)
glocalisations = tf.concat(fglocalisations, axis=0)
dtype = logits.dtype
# 计算正类掩膜
pmask = gscores > match_threshold
fpmask = tf.cast(pmask, dtype)
n_positives = tf.reduce_sum(fpmask)
# 硬负例挖掘
no_classes = tf.cast(pmask, tf.int32)
predictions = slim.softmax(logits)
nmask = tf.logical_and(tf.logical_not(pmask), gscores > -0.5)
fnmask = tf.cast(nmask, dtype)
nvalues = tf.where(nmask, predictions[:, 0], 1. - fnmask)
nvalues_flat = tf.reshape(nvalues, [-1])
# 要选择的负例的数目
max_neg_entries = tf.cast(tf.reduce_sum(fnmask), tf.int32)
n_neg = tf.cast(negative_ratio * n_positives, tf.int32) + batch_size
n_neg = tf.minimum(n_neg, max_neg_entries)

val, idxes = tf.nn.top_k(-nvalues_flat, k=n_neg)
max_hard_pred = -val[-1]
# 最终的负类掩膜
nmask = tf.logical_and(nmask, nvalues < max_hard_pred)
fnmask = tf.cast(nmask, dtype)
# 增加交叉熵损失
with tf.name_scope('cross_entropy_pos'):
    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits,
        labels=gclasses)
    loss = tf.div(tf.reduce_sum(loss * fpmask), batch_size, name='value')
    tf.losses.add_loss(loss)
with tf.name_scope('cross_entropy_neg'):
    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits,
        labels=no_classes)
    loss = tf.div(tf.reduce_sum(loss * fnmask), batch_size, name='value')
    tf.losses.add_loss(loss)
# 添加位置损失：平滑 L1、L2
with tf.name_scope('localization'):
    # 权值张量：正类掩膜 + 随机负类掩膜
    weights = tf.expand_dims(alpha * fpmask, axis=-1)
    loss = custom_layers.abs_smooth(localisations - glocalisations)
    loss = tf.div(tf.reduce_sum(loss * weights), batch_size, name='value')
    tf.losses.add_loss(loss)
def ssd_losses_old(logits, localisations, gclasses, glocalisations, gscores, match_
threshold=0.5, negative_ratio=3., alpha=1., label_smoothing=0., device='/cpu:0',
scope=None):
# 训练基于 VGGNet 网络的 SSD 300 的损失函数

```



```

with tf.device(device):
    with tf.name_scope(scope, 'ssd_losses'):
        l_cross_pos = []
        l_cross_neg = []
        l_loc = []
        for i in range(len(logits)):
            dtype = logits[i].dtype
            with tf.name_scope('block_%i' % i):
                # 调整权值
                wsize = tfe.get_shape(logits[i], rank=5)
                wsize = wsize[1] * wsize[2] * wsize[3]
                # 正类掩膜
                pmask = gscores[i] > match_threshold
                fpmask = tf.cast(pmask, dtype)
                n_positives = tf.reduce_sum(fpmask)
                # 负类掩膜
                no_classes = tf.cast(pmask, tf.int32)
                predictions = slim.softmax(logits[i])
                nmask = tf.logical_and(tf.logical_not(pmask), gscores[i] > -0.5)
                fnmask = tf.cast(nmask, dtype)
                nvalues = tf.where(nmask, predictions[:, :, :, :, 0], 1. - fnmask)
                nvalues_flat = tf.reshape(nvalues, [-1])
                # 选择的负例的数目
                n_neg = tf.cast(negative_ratio * n_positives, tf.int32)
                n_neg = tf.maximum(n_neg, tf.size(nvalues_flat) // 8)
                n_neg = tf.maximum(n_neg, tf.shape(nvalues)[0] * 4)
                max_neg_entries = 1 + tf.cast(tf.reduce_sum(fnmask), tf.int32)
                n_neg = tf.minimum(n_neg, max_neg_entries)
                val, idxes = tf.nn.top_k(-nvalues_flat, k=n_neg)
                max_hard_pred = -val[-1]
                # 最终的负类掩膜
                nmask = tf.logical_and(nmask, nvalues < max_hard_pred)
                fnmask = tf.cast(nmask, dtype)
                with tf.name_scope('cross_entropy_pos'): # 添加交叉熵损失
                    fpmask = wsize * fpmask
                    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits[i], labels=gclasses[i])
                    loss = tf.losses.compute_weighted_loss(loss, fpmask)
                    l_cross_pos.append(loss)
                with tf.name_scope('cross_entropy_neg'):
                    fnmask = wsize * fnmask
                    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits[i], labels=no_classes)
                    loss = tf.losses.compute_weighted_loss(loss, fnmask)
                    l_cross_neg.append(loss)

            with tf.name_scope('localization'): # 添加位置损失: 平滑 L1、L2
                # 权值张量: 正类掩膜 + 随机掩膜
                weights = tf.expand_dims(alpha * fpmask, axis=-1)

```

```

        loss = custom_layers.abs_smooth(localisations[i] - gloca-
        lisations[i])
        loss = tf.losses.compute_weighted_loss(loss, weights)
        l_loc.append(loss)

    with tf.name_scope('total'):
        # 另外的全部损失
        total_cross_pos = tf.add_n(l_cross_pos, 'cross_entropy_pos')
        total_cross_neg = tf.add_n(l_cross_neg, 'cross_entropy_neg')
        total_cross = tf.add(total_cross_pos, total_cross_neg, 'cross-
        entropy')
        total_loc = tf.add_n(l_loc, 'localization')

    # 将损失添加到 EXTRA LOSSES 中
    tf.add_to_collection('EXTRA_LOSSES', total_cross_pos)
    tf.add_to_collection('EXTRA_LOSSES', total_cross_neg)
    tf.add_to_collection('EXTRA_LOSSES', total_cross)
    tf.add_to_collection('EXTRA_LOSSES', total_loc)

```

8.5.3 SSD 的图像目标检测案例及演示效果

本节描述一个利用 SSD 在 TensorFlow 框架下进行图像目标检测的案例，其中用到的 VOC 2007 图像数据集可以根据表 1.2 提供的地址下载。

SSD 网络的训练步骤为：①参照图 8.21 的命令，把训练集和验证集转换成 TensorFlow 的图像格式；②参照图 8.22，执行训练命令。如图 8.23 所示，在训练到 119 350 次时，SSD 网络的损失为 79.0067，在训练到 119 355 次时保存一次中间结果。如图 8.24 所示，在训练到 120 000 次时结束，保存结果，此时 SSD 网络的损失为 53.9744。

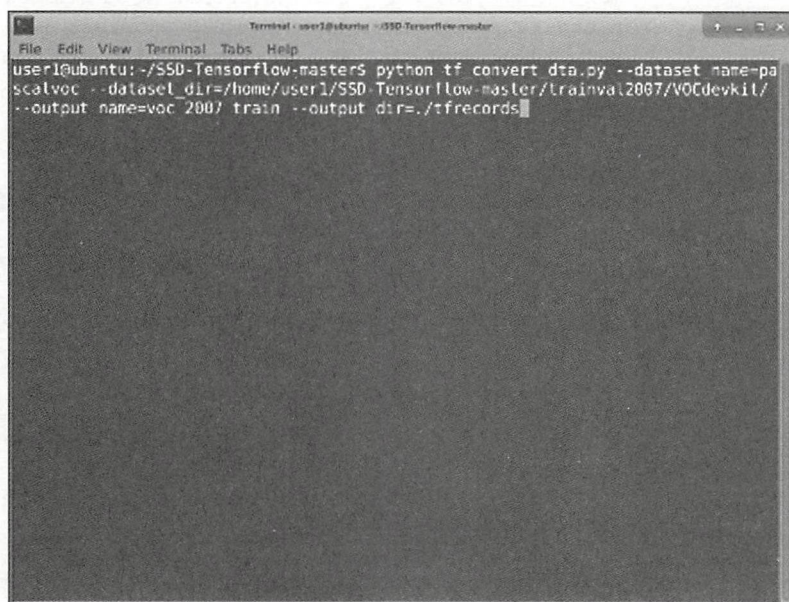
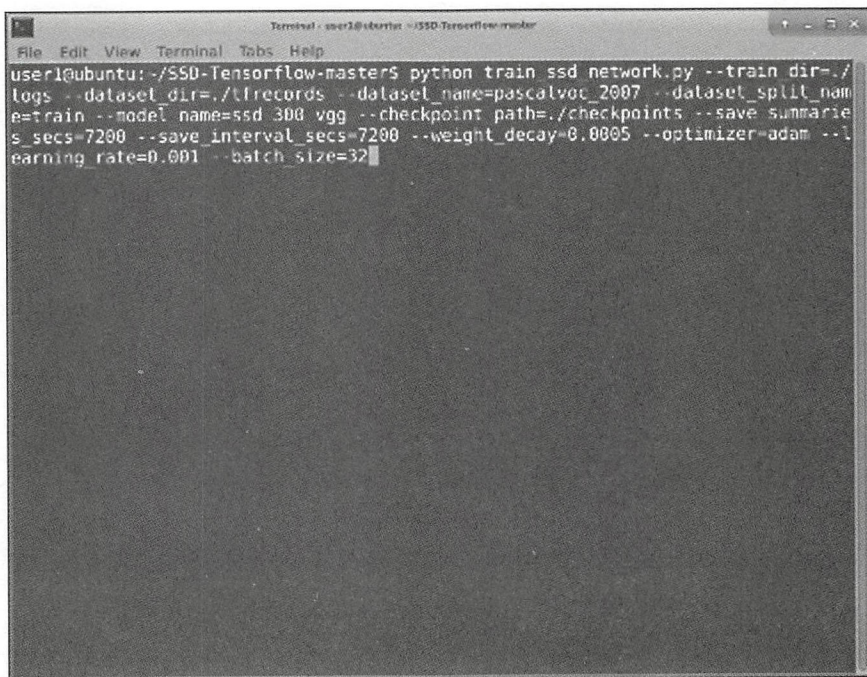
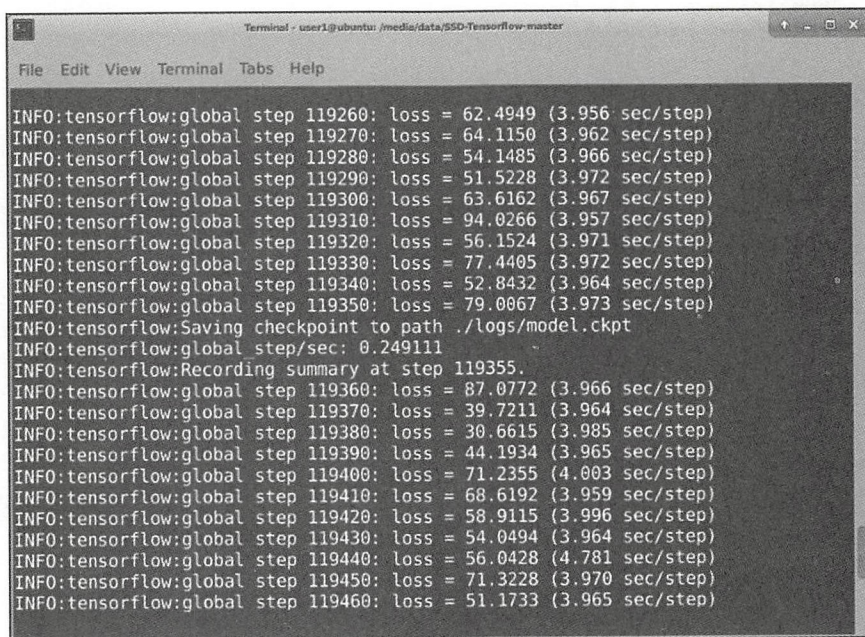


图 8.21 SSD 对训练集和验证集的格式转换命令



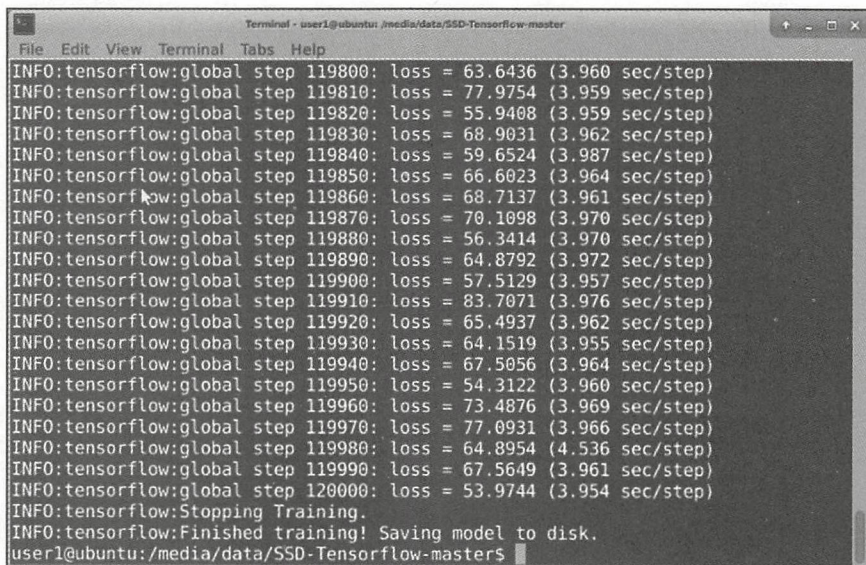
```
Terminal - user1@ubuntu: ~/SSD-Tensorflow-master
File Edit View Terminal Tabs Help
user1@ubuntu:~/SSD-Tensorflow-master$ python train_ssd_network.py --train_dir=./
logs --dataset_dir=./tfrecords --dataset_name=pascal3d+2007 --dataset_split_name=train --model_name=ssd_300_vgg --checkpoint_path=./checkpoints --save_summaries_secs=7200 --save_interval_secs=7200 --weight_decay=0.0005 --optimizer=adam --learning_rate=0.001 --batch_size=32
```

图 8.22 SSD 图像目标检测案例程序的训练命令



```
Terminal - user1@ubuntu: ~/media/data/SSD-Tensorflow-master
File Edit View Terminal Tabs Help
INFO:tensorflow:global step 119260: loss = 62.4949 (3.956 sec/step)
INFO:tensorflow:global step 119270: loss = 64.1150 (3.962 sec/step)
INFO:tensorflow:global step 119280: loss = 54.1485 (3.966 sec/step)
INFO:tensorflow:global step 119290: loss = 51.5228 (3.972 sec/step)
INFO:tensorflow:global step 119300: loss = 63.6162 (3.967 sec/step)
INFO:tensorflow:global step 119310: loss = 94.0266 (3.957 sec/step)
INFO:tensorflow:global step 119320: loss = 56.1524 (3.971 sec/step)
INFO:tensorflow:global step 119330: loss = 77.4405 (3.972 sec/step)
INFO:tensorflow:global step 119340: loss = 52.8432 (3.964 sec/step)
INFO:tensorflow:global step 119350: loss = 79.0067 (3.973 sec/step)
INFO:tensorflow:Saving checkpoint to path ./logs/model.ckpt
INFO:tensorflow:global step/sec: 0.249111
INFO:tensorflow:Recording summary at step 119355.
INFO:tensorflow:global step 119360: loss = 87.0772 (3.966 sec/step)
INFO:tensorflow:global step 119370: loss = 39.7211 (3.964 sec/step)
INFO:tensorflow:global step 119380: loss = 30.6615 (3.985 sec/step)
INFO:tensorflow:global step 119390: loss = 44.1934 (3.965 sec/step)
INFO:tensorflow:global step 119400: loss = 71.2355 (4.003 sec/step)
INFO:tensorflow:global step 119410: loss = 68.6192 (3.959 sec/step)
INFO:tensorflow:global step 119420: loss = 58.9115 (3.996 sec/step)
INFO:tensorflow:global step 119430: loss = 54.0494 (3.964 sec/step)
INFO:tensorflow:global step 119440: loss = 56.0428 (4.781 sec/step)
INFO:tensorflow:global step 119450: loss = 71.3228 (3.970 sec/step)
INFO:tensorflow:global step 119460: loss = 51.1733 (3.965 sec/step)
```

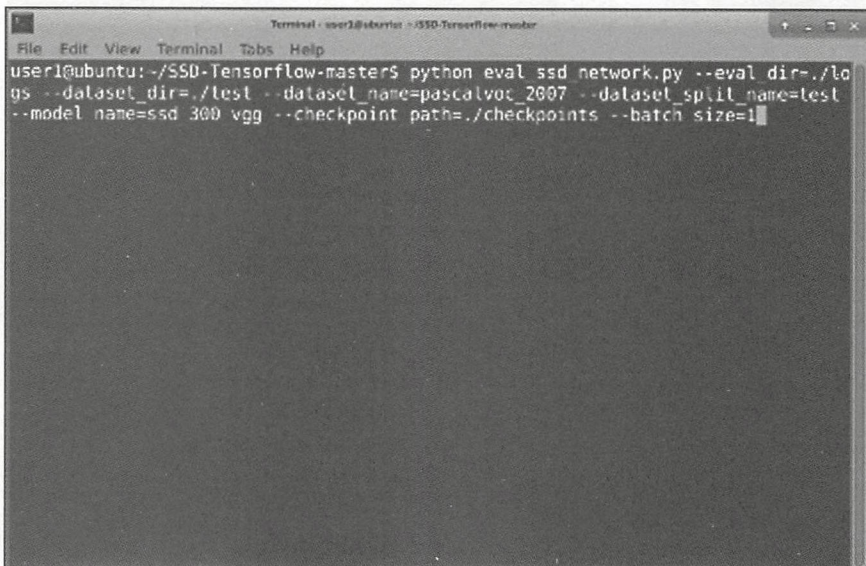
图 8.23 SSD 图像目标检测案例程序在训练到 119 260 次至 119 460 次的中间信息



```
Terminal - user1@ubuntu: /media/data/SSD-Tensorflow-master
File Edit View Terminal Tabs Help
INFO:tensorflow:global step 119800: loss = 63.6436 (3.960 sec/step)
INFO:tensorflow:global step 119810: loss = 77.9754 (3.959 sec/step)
INFO:tensorflow:global step 119820: loss = 55.9408 (3.959 sec/step)
INFO:tensorflow:global step 119830: loss = 68.9031 (3.962 sec/step)
INFO:tensorflow:global step 119840: loss = 59.6524 (3.987 sec/step)
INFO:tensorflow:global step 119850: loss = 66.6023 (3.964 sec/step)
INFO:tensorflow:global step 119860: loss = 68.7137 (3.961 sec/step)
INFO:tensorflow:global step 119870: loss = 70.1098 (3.970 sec/step)
INFO:tensorflow:global step 119880: loss = 56.3414 (3.970 sec/step)
INFO:tensorflow:global step 119890: loss = 64.8792 (3.972 sec/step)
INFO:tensorflow:global step 119900: loss = 57.5129 (3.957 sec/step)
INFO:tensorflow:global step 119910: loss = 83.7071 (3.976 sec/step)
INFO:tensorflow:global step 119920: loss = 65.4937 (3.962 sec/step)
INFO:tensorflow:global step 119930: loss = 64.1519 (3.955 sec/step)
INFO:tensorflow:global step 119940: loss = 67.5056 (3.964 sec/step)
INFO:tensorflow:global step 119950: loss = 54.3122 (3.960 sec/step)
INFO:tensorflow:global step 119960: loss = 73.4876 (3.969 sec/step)
INFO:tensorflow:global step 119970: loss = 77.0931 (3.966 sec/step)
INFO:tensorflow:global step 119980: loss = 64.8954 (4.536 sec/step)
INFO:tensorflow:global step 119990: loss = 67.5649 (3.961 sec/step)
INFO:tensorflow:global step 120000: loss = 53.9744 (3.954 sec/step)
INFO:tensorflow:Stopping Training.
INFO:tensorflow:Finished training! Saving model to disk.
user1@ubuntu: /media/data/SSD-Tensorflow-master$
```

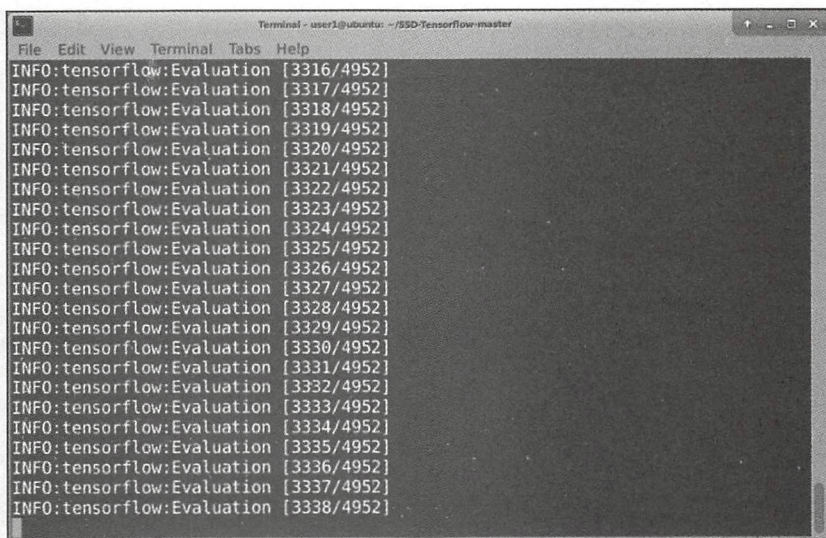
图 8.24 SSD 图像目标检测案例程序在训练结束时的结果信息

SSD 网络的测试命令如图 8.25 所示，测试信息如图 8.26 所示，测试结果如图 8.27 所示。从图 8.27 中可以看出，在 VOC 2007 的测试集上，SSD 的平均准确率约为 59.93%，运行时间为 1147.332 秒，平均每幅图像花费 0.232 秒。注意，测试前也要参照图 8.21 对测试集进行格式转换。



```
Terminal - user1@ubuntu: /SSD-Tensorflow-master
File Edit View Terminal Tabs Help
user1@ubuntu:~/SSD-Tensorflow-master$ python eval_ssd_network.py --eval_dir=./logs
--dataset_dir=./test --dataset_name=pascal_voc_2007 --dataset_split_name=test
--model_name=ssd_300_vgg --checkpoint_path=./checkpoints --batch_size=1
```

图 8.25 SSD 图像目标检测案例程序的测试命令

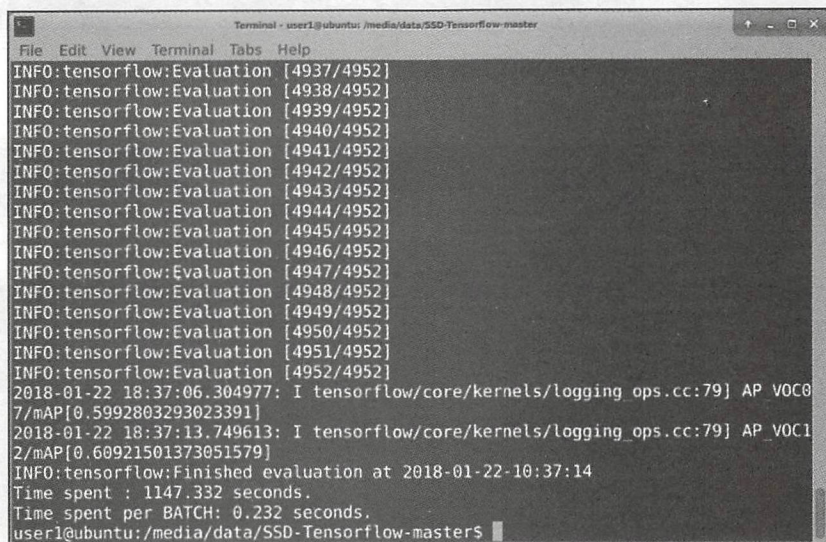


```

Terminal - user1@ubuntu: ~/SSD-Tensorflow-master
File Edit View Terminal Tabs Help
INFO:tensorflow:Evaluation [3316/4952]
INFO:tensorflow:Evaluation [3317/4952]
INFO:tensorflow:Evaluation [3318/4952]
INFO:tensorflow:Evaluation [3319/4952]
INFO:tensorflow:Evaluation [3320/4952]
INFO:tensorflow:Evaluation [3321/4952]
INFO:tensorflow:Evaluation [3322/4952]
INFO:tensorflow:Evaluation [3323/4952]
INFO:tensorflow:Evaluation [3324/4952]
INFO:tensorflow:Evaluation [3325/4952]
INFO:tensorflow:Evaluation [3326/4952]
INFO:tensorflow:Evaluation [3327/4952]
INFO:tensorflow:Evaluation [3328/4952]
INFO:tensorflow:Evaluation [3329/4952]
INFO:tensorflow:Evaluation [3330/4952]
INFO:tensorflow:Evaluation [3331/4952]
INFO:tensorflow:Evaluation [3332/4952]
INFO:tensorflow:Evaluation [3333/4952]
INFO:tensorflow:Evaluation [3334/4952]
INFO:tensorflow:Evaluation [3335/4952]
INFO:tensorflow:Evaluation [3336/4952]
INFO:tensorflow:Evaluation [3337/4952]
INFO:tensorflow:Evaluation [3338/4952]

```

图 8.26 SSD 图像目标检测案例程序的测试信息



```

Terminal - user1@ubuntu: /media/data/SSD-Tensorflow-master
File Edit View Terminal Tabs Help
INFO:tensorflow:Evaluation [4937/4952]
INFO:tensorflow:Evaluation [4938/4952]
INFO:tensorflow:Evaluation [4939/4952]
INFO:tensorflow:Evaluation [4940/4952]
INFO:tensorflow:Evaluation [4941/4952]
INFO:tensorflow:Evaluation [4942/4952]
INFO:tensorflow:Evaluation [4943/4952]
INFO:tensorflow:Evaluation [4944/4952]
INFO:tensorflow:Evaluation [4945/4952]
INFO:tensorflow:Evaluation [4946/4952]
INFO:tensorflow:Evaluation [4947/4952]
INFO:tensorflow:Evaluation [4948/4952]
INFO:tensorflow:Evaluation [4949/4952]
INFO:tensorflow:Evaluation [4950/4952]
INFO:tensorflow:Evaluation [4951/4952]
INFO:tensorflow:Evaluation [4952/4952]
2018-01-22 18:37:06.304977: I tensorflow/core/kernels/logging_ops.cc:79] AP_VOC0
7/mAP[0.5992803293023391]
2018-01-22 18:37:13.749613: I tensorflow/core/kernels/logging_ops.cc:79] AP_VOC1
2/mAP[0.60921501373051579]
INFO:tensorflow:Finished evaluation at 2018-01-22-10:37:14
Time spent : 1147.332 seconds.
Time spent per BATCH: 0.232 seconds.
user1@ubuntu: /media/data/SSD-Tensorflow-masters

```

图 8.27 SSD 图像目标检测案例程序在测试结束时的结果信息

最后，测试结果还能够可视化，具体步骤为：①参照图 8.28 的命令启动程序；②在出现图 8.29 所示的 Jupyter 界面后，单击运行按钮，随即产生一幅原始图像的可视化检测结果，如图 8.30 所示。注意，图 8.30 的可视化结果只给出了物体的类别编号和置信度。如果需要显示类别名字，可根据方框 8.3 和 8.4 修改可视化程序，主要修改之处是在原始 visualization.py 文件的第 89 行之前增加方框 8.3 的代码，并把第 109 行替换为方框 8.4 的代码。修改之后的可视化结果如图 8.31 所示。

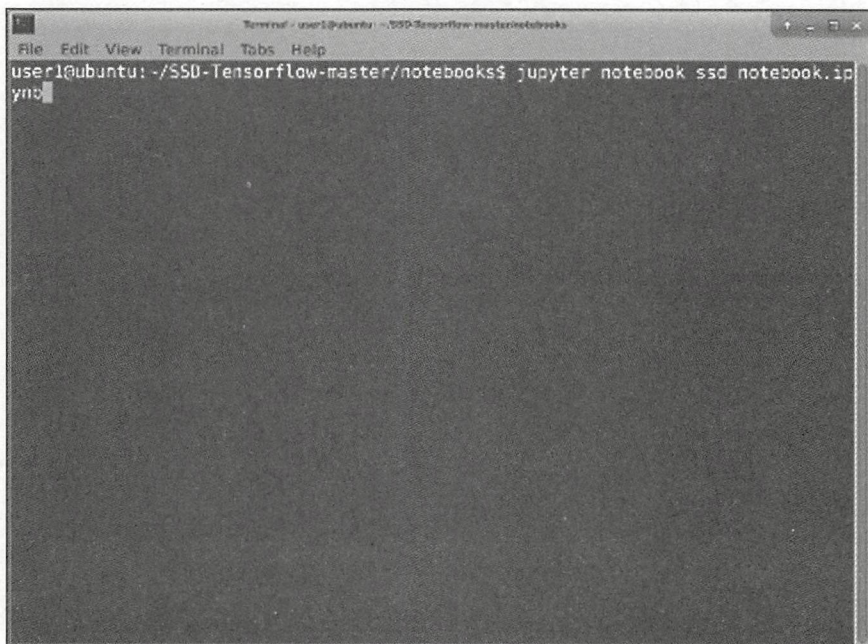


图 8.28 SSD 图像目标检测案例程序的可视化命令

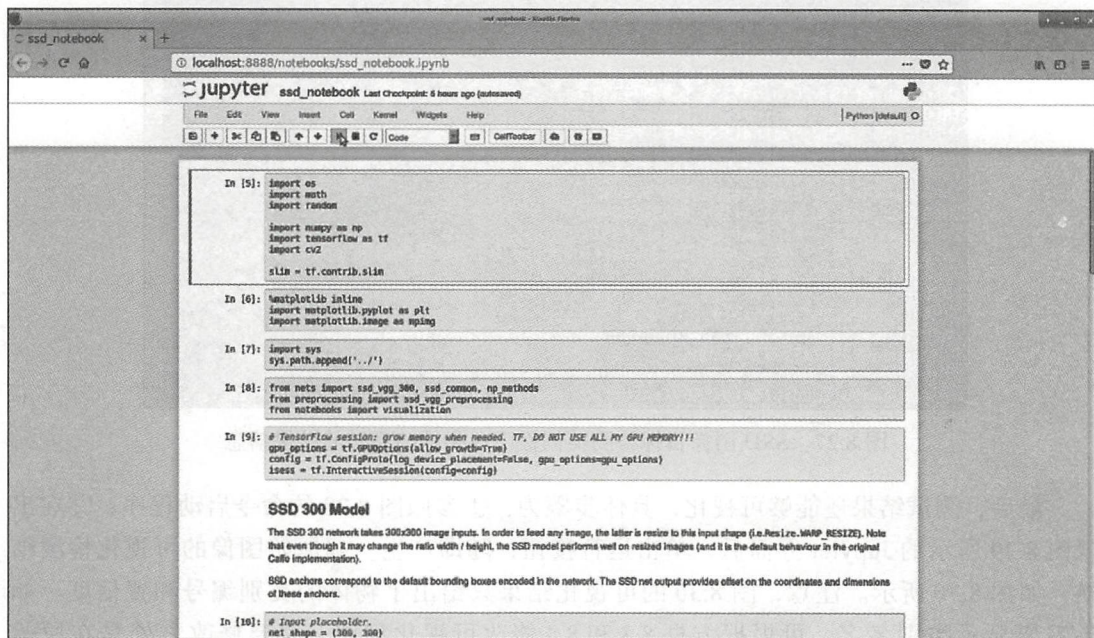


图 8.29 SSD 图像目标检测案例程序的 Jupyter 编辑器界面

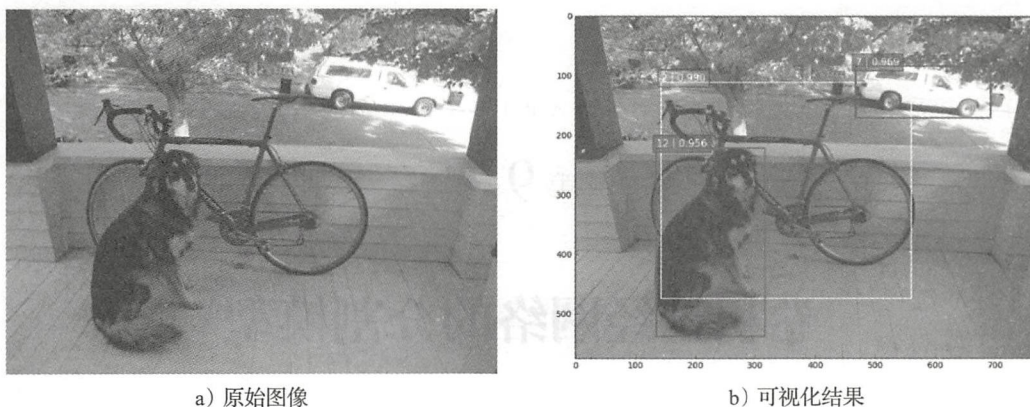


图 8.30 SSD 图像目标检测案例程序

方框 8.3 visualization.py 文件中增加的代码

```
def num2class(n):
    from datasets import pascalvoc_2007 as pas
    x=pas.pascalvoc_common.VOC_LABELS.items()
    for name,item in x:
        if n in item:
            return name
```

方框 8.4 visualization.py 文件中替换的代码

```
class_name = num2class(cls_id)
```

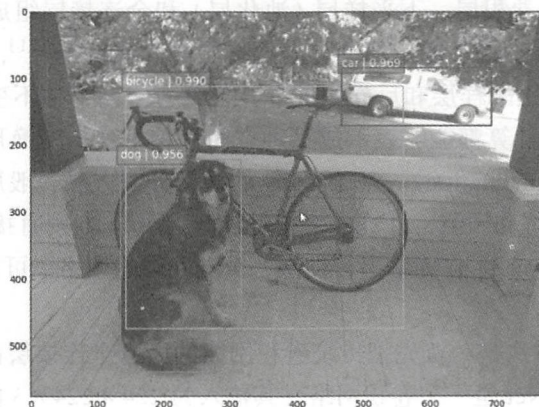


图 8.31 SSD 图像目标检测案例程序的修改可视化结果

卷积神经网络的分割模型

随着卷积神经网络在目标检测任务上的推进，它也开始被应用于更精细的图像处理任务：语义分割和实例分割。目标检测只需预测图像中每个对象的位置和类别，语义分割还要把每个像素都进行分类，而实例分割的任务则更难，要进一步把每个对象的不同实例都区分开来。本章主要介绍卷积神经网络的3种分割模型，包括全卷积网络、金字塔场景分析网络和掩膜区域卷积网络，以及它们的Caffe或TensorFlow代码实现、图像分割案例和演示效果。

9.1 全卷积网络 FCN

9.1.1 FCN 的模型结构

通常，卷积网络由卷积层、下采样层（池化层）和全连接层组成。全卷积网络（Fully Convolutional Network, FCN）是一种没有全连接层的卷积网络^[21]，但除了卷积层和下采样层，另外还可以包含上采样层和反卷积层等其他具有空间平移不变形式的层。FCN的关键特征在于其所有层的计算都能够表示成某种空间平移不变的变换形式，主要用于图像的语义分割（semantic segmentation）。基于CNN的语义分割方法一般用每个像素周围的图像块作为输入进行分类训练和预测，计算效率相对较低。而FCN则直接把整幅图像作为输入、把人工标签地图（label map）作为输出，训练一个端到端的网络，可以显著提高语义分割的计算效率和预测性能。

一种构造FCN的方法是，首先把传统卷积网络的所有全连接层都改造成相应大小的密集卷积层。比如，AlexNet是一个卷积网络，其输入为 $224 \times 224 \times 3$ 的图像，经过一系列的卷积层和池化层后，得到一个大小为 $7 \times 7 \times 512$ 的卷积层，再经过两个含有4096个节点的全连接层，产生一个有1000个节点的输出层。相应的FCN把第1个含有4096个的节点的全连接层改造成4096个卷积核大小为 7×7 的 1×1 卷积面，把第2个含有4096个节点的全连接层改造成4096个卷积核大小为 1×1 的 1×1 卷积面，把1000个节点的输出层改造成1000个卷积核大小为 1×1 的 1×1 卷积面（这个卷积面称为热力图）。改造过程如图9.1

所示,从中可以看出,FCN 的输入图像大小为 $224 \times 224 \times 3$,产生的热力图是 1000 个点,但在输入图像更大时热力图则是 1000 个面(即二维数组)。

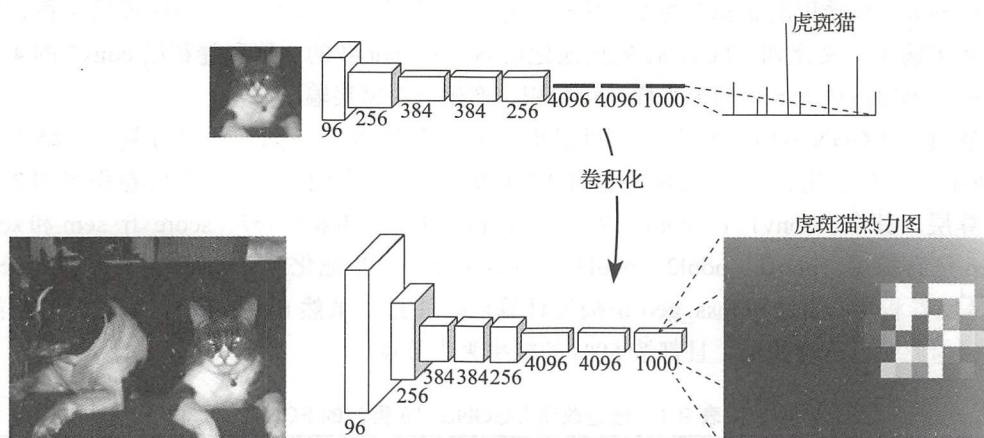


图 9.1 FCN 把 AlexNet 的全连接层改造成卷积层的过程,其中输出层被改造成热力图

在改造全连接层之后,FCN 有两种方式产生密集输出:直接放大和拼接放大。直接放大是通过放大变换(比如上采样和反卷积),直接把热力图放大成一个与输入大小相同的分割面。比如,在图 9.2 中,FCN-32s 直接把卷积层 conv7 通过放大 32 倍的上采样或反卷积,

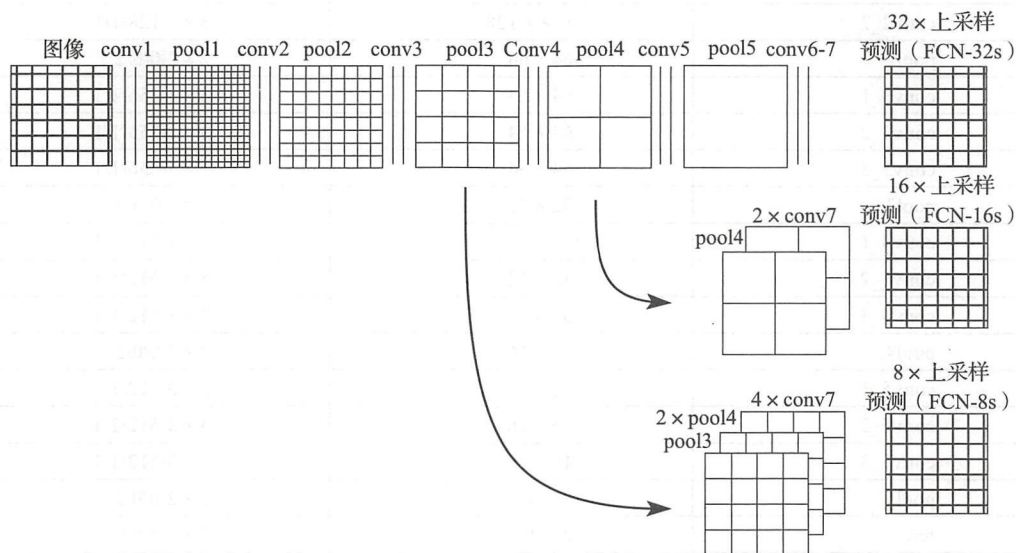


图 9.2 FCN 产生密集输出的方式:直接放大和拼接放大。池化层和预测层使用相对粗糙的空间网格表示,而卷积层使用垂直线表示。第一行的 FCN-32s 是单流(single stream)网络,通过步长为 32 的上采样进行单步像素预测。第二行的 FCN-16s 结合最后一层和 pool4 层的预测,通过步长为 16 的上采样进行精细的预测,并保持高层语义信息。第三行的 FCN-8s 还另外拼接了 pool3 层,通过步长为 8 的上采样提供更精细的预测

产生一个密集输出 (dense output)。拼接放大是使用跨层连接 (skip connection) 将不同粗细粒度的信息先进行拼接再放大, 最后产生一个密集输出。比如, 在图 9.2 中, FCN-16s 先把池化层 pool4 和卷积层 conv7 的 2 倍进行拼接, 再通过放大 16 倍的上采样或反卷积, 产生一个密集输出。又比如, FCN-8s 先把池化层 pool3、pool4 的 2 倍和卷积层 conv7 的 4 倍进行拼接, 再通过放大 8 倍的上采样或反卷积, 产生一个密集输出。

通过对 VGGNet-19 进行改造, 可以得到另一个 FCN 进行图像语义分割, 其结构如表 9.1 所示。不难看出, 这个 FCN 共包含 17 个卷积层、5 个池化层、2 个反卷积层和 2 个损失计算层。其中, conv1_1、conv1_2、...、conv5_3 以及 fc6 和 fc7、score_fr_sem 和 score_fr_geo 是卷积层, pool1、pool2、pool3、pool4 和 pool5 是池化层, upscore_sem 和 upscore_geo 是反卷积层, loss 和 loss_geo 是损失计算层。注意, 虽然 fc6 和 fc7 保留了全连接层的名字, 但此时已是卷积层, 且都按 50% 的概率丢失节点。

表 9.1 通过改造 VGGNet-19 得到的 FCN

layer name	output size	kernel size/number/pad/stride
Input	256×256	$256 \times 256 \times 3$
conv1_1	256×256	$3 \times 3/64/1/1$
conv1_2	256×256	$3 \times 3/64/1/1$
pool1	128×128	$2 \times 2/0/0/2$
conv2_1	128×128	$3 \times 3/128/1/1$
conv2_2	128×128	$3 \times 3/128/1/1$
pool2	64×64	$2 \times 2/0/0/2$
conv3_1	64×64	$3 \times 3/256/1/1$
conv3_2	64×64	$3 \times 3/256/1/1$
conv3_3	64×64	$3 \times 3/256/1/1$
pool3	32×32	$2 \times 2/0/0/2$
conv4_1	32×32	$3 \times 3/512/1/1$
conv4_2	32×32	$3 \times 3/512/1/1$
conv4_3	32×32	$3 \times 3/512/1/1$
pool4	16×16	$2 \times 2/0/0/2$
conv5_1	16×16	$3 \times 3/512/1/1$
conv5_2	16×16	$3 \times 3/512/1/1$
conv5_3	16×16	$3 \times 3/512/1/1$
pool5	8×8	$2 \times 2/0/0/2$
fc6	2×2	$7 \times 7/4096/0/1$
fc7	2×2	$7 \times 7/4096/0/1$
score_fr_sem	2×2	$1 \times 1/33/0/1$
upscore_sem		$64 \times 64/33/0/3$
score_sem		2/19
loss		

(续)

layer name	output size	kernel size/number/pad/stride
score_fr_geo		$1 \times 1/3/0/1$
upscpre_geo		$64 \times 64/3/0/32$
score_geo		$2/19$
loss_geo		

9.1.2 FCN 的 Caffe 代码实现及说明

从网址 <https://github.com/shelhamer/fcn.berkeleyvision.org> 可以下载一套 FCN 的 Caffe 实现代码。这套代码是基于表 9.1 实现的, 用来对 SIFT Flow 图像数据集进行语义分割, 包含 3 个子文件夹, 分别为 siftflow-fcn8s、siftflow-fcn16s 和 siftflow-fcn32s。这 3 个子文件夹的内容大致相似, 都包含网络结构文件 trainval.prototxt 和 test.prototxt、求解器配置文件 solver.prototxt, 以及 Python 接口文件 solve.py。下面基于 siftflow-fcn32s 的 trainval.prototxt 文件, 对 FCN 的几个关键部分进行说明, 包括数据层 data、卷积层 conv1_1、最大池化层 pool1、分割层 score_fr_sem、分割损失层 loss、几何层 score_fr_geo 和几何损失 loss_geo。

1. 数据层 data 的实现代码及说明

```
layer {
  name: "data"                # 表示该层是数据层
  type: "Python"              # 表示该层由 Python 定义
  top: "data"
  top: "sem"                  # 表示语义类别
  top: "geo"                  # 表示几何类别
  python_param {
    module: "siftflow_layers"  # 表示模块的名字
    layer: "SIFTFlowSegDataLayer" # 表示层的名字, 即模块中的类名
    param_str: "{\'siftflow_dir\': \'../data/sift-flow\', \'seed\': 1337, \'split\': \'trainval\'}"
  }
}
```

2. 卷积层 conv1_1 的实现代码及说明

```
layer {
  name: "conv1_1"
  type: "Convolution"        # 表示该层为卷积层
  bottom: "data"              # 底层为输入数据
  top: "conv1_1"
  param {
    lr_mult: 1                # 权值的学习率系数
    decay_mult: 1              # 权值的衰减系数
  }
  param {
    lr_mult: 2                # 偏置的学习率系数
  }
}
```

```

        decay_mult: 0
    }
    convolution_param {
        num_output: 64          # 表示输出特征图的个数
        pad: 100                # 对输入的4边各扩展100个单位长度，并用0填充
        kernel_size: 3          # 卷积核的大小
        stride: 1                # 卷积核的步长
    }
}
layer {
    name: "relu1_1"
    type: "ReLU"                # 把激活函数选为 ReLU
    bottom: "conv1_1"
    top: "conv1_1"
}

```

3. 最大池化层 pool1 的实现代码及说明

```

layer {
    name: "pool1"
    type: "Pooling"             # 表示该层为池化层
    bottom: "conv1_2"
    top: "pool1"
    pooling_param {
        pool: MAX                # 把池化方式选为最大池化
        kernel_size: 2           # 池化窗口的大小
        stride: 2                # 池化窗口的步长
    }
}

```

4. 分割层 score_fr_sem 的实现代码及说明

```

layer {
    name: "score_fr_sem"
    type: "Convolution"         # 表示该层为卷积层
    bottom: "fc7"               # 注意，底层虽然是卷积层，但仍以全连接层命名
    top: "score_fr_sem"
    param {
        lr_mult: 1              # 卷积核的学习率系数
        decay_mult: 1           # 卷积核的权值衰减系数
    }
    param {
        lr_mult: 2              # 偏置的学习率系数
        decay_mult: 0           # 偏置的权值衰减系数
    }
    convolution_param {
        num_output: 33          # 输出特征图的个数
        pad: 0                  # 对输入的4边不进行扩展填充
        kernel_size: 1          # 卷积核的大小
    }
}
layer {
    name: "upscore_sem"

```



```

type: "Deconvolution"                # 表示反卷积层
bottom: "score_fr_sem"
top: "upscore_sem"
param {
    lr_mult: 0                        # 表示卷积核的学习率系数
}
convolution_param {
    num_output: 33                    # 输出特征图的个数
    bias_term: false                  # 不学习偏置
    kernel_size: 64                   # 卷积核的大小
    stride: 32                        # 卷积核的步长
}
}
layer {
    name: "score_sem"
    type: "Crop"                      # 表示该层为裁剪层
    bottom: "upscore_sem"             # 底层为 upscore_sem 层, 表示要进行裁剪的层
    bottom: "data"                    # 底层为 data 层, 表示裁剪时参考的对象层
    top: "score_sem"
    crop_param {
        axis: 2                       # 表示裁剪轴 2 及其之后的所有轴
        offset: 19                    # 表示裁剪的开始像素
    }
}
}

```

5. 分割损失层 loss 的代码实现及说明

```

layer {
    name: "loss"
    type: "SoftmaxWithLoss"           # 表示带损失的软最大输出层
    bottom: "score_sem"
    bottom: "sem"
    top: "loss"
    loss_param {
        ignore_label: 255             # 表示标签为 255 的类别不参与损失计算
        normalize: false              # 表示不进行归一化
    }
}
}

```

6. 几何层 score_fr_geo 的代码实现及说明

```

layer {
    name: "score_fr_geo"
    type: "Convolution"               # 表示该层为卷积层
    bottom: "fc7"
    top: "score_fr_geo"
    param {
        lr_mult: 1                    # 表示卷积核的学习率系数
        decay_mult: 1                 # 表示卷积核的权值衰减系数
    }
    param {
        lr_mult: 2                    # 表示偏置的学习率系数
        decay_mult: 0                 # 表示偏置的权值衰减系数
    }
}

```

```

    }
    convolution_param {
        num_output: 3                # 表示输出特征图的个数
        pad: 0
        kernel_size: 1              # 表示卷积核的大小
    }
}
layer {
    name: "upscore_geo"
    type: "Deconvolution"           # 表示该层为反卷积层
    bottom: "score_fr_geo"
    top: "upscore_geo"
    param {
        lr_mult: 0
    }
    convolution_param {
        num_output: 3              # 表示输出特征图的个数
        bias_term: false           # 表示不学习偏置
        kernel_size: 64            # 表示卷积核的大小
        stride: 32                  # 表示卷积核步长
    }
}
}
layer {
    name: "score_geo"
    type: "Crop"
    bottom: "upscore_geo"
    bottom: "data"
    top: "score_geo"
    crop_param {
        axis: 2
        offset: 19
    }
}
}

```

7. 几何损失 loss_geo 的代码实现及说明

```

layer {
    name: "loss_geo"
    type: "SoftmaxWithLoss"        # 表示带损失的软最大输出层
    bottom: "score_geo"
    bottom: "geo"
    top: "loss_geo"
    loss_param {
        ignore_label: 255
        normalize: false
    }
}
}

```

9.1.3 FCN 的图像语义和几何分割案例

本节描述一个利用 FCN 在 Caffe 框架下进行图像语义分割的案例，其中用到的 SIFT Flow 数据集可以根据表 1.2 提供的地址下载。在 SIFT Flow 中，图像的像素有 33 个语义

类别标记（桥、山、太阳等）和3个几何类别标记（水平、竖直和天空）。采用全卷积网络 FCN-32s 的双头版（two-headed version），可以通过学习这两种语义的联合表示来同时预测它们。该网络具有语义预测层、几何预测层及相应的损失。

语义分割精度的常用衡量标准包括：像素精度（Pixel Accuracy, PA）、平均像素精度（Mean Pixel Accuracy, MPA）、平均交并比（Mean Intersection over Union, MIoU）和频权交并比（Frequency Weighted Intersection over Union, FWIoU）。下面进行逐一说明。假设共有 $k+1$ 个类， p_{ij} 表示属于第 i 类但被预测为第 j 类的像素数量。

像素精度：这是最简单的度量，表示标记正确的像素占总像素的比例，其计算公式为

$$PA = \frac{\sum_{i=0}^k p_{ii}}{\sum_{i=0}^k \sum_{j=0}^k p_{ij}}$$

平均像素精度（MPA）：是像素精度的一种简单提升，计算每个类内被正确分类的像素数的比例，之后求所有类的平均，即

$$MPA = \frac{1}{k+1} \sum_{i=0}^k \frac{p_{ii}}{\sum_{j=0}^k p_{ij}}$$

平均交并比：是语义分割的标准度量，表示计算真实值（ground truth）和预测值（predicted segmentation）的交并比之平均值，即

$$MIoU = \frac{1}{k+1} \sum_{i=0}^k \frac{p_{ii}}{\sum_{j=0}^k p_{ij} + \sum_{j=0}^k p_{ij} - p_{ii}}$$

频权并交比：表示根据每个类出现的频率来设置其权重，计算公式为

$$FWIoU = \frac{1}{\sum_{i=0}^k \sum_{j=0}^k p_{ij}} \sum_{i=0}^k \frac{p_{ii}}{\sum_{j=0}^k p_{ij} + \sum_{j=0}^k p_{ij} - p_{ii}}$$

利用 FCN 对 SIFT Flow 进行图像语义分割，还需要按照方框 9.1 修改求解器配置文件 solver.prototxt，并按照方框 9.2 修改 Python 接口文件 solve.py。注意：程序启动时，可能会出现“No module name surgery”的问题，解决办法是把 fcn.berkeleyvision.org-master 文件夹下的 surgery.py 拷贝到 siftflow-fcn32s 文件夹。

方框 9.1 在 solver.prototxt 中设置的超参数情况

```
train_net: "trainval.prototxt"
test_net: "test.prototxt"
test_iter: 100
# make test net, but don't invoke it from the solver itself
test_interval: 999999999
display: 20
average_loss: 20
```



```

lr_policy: "fixed"
# lr for unnormalized softmax
base_lr: 1e-10
# high momentum
momentum: 0.99
# no gradient accumulation
iter_size: 1
max_iter: 100000
weight_decay: 0.0005
snapshot: 1000
snapshot_prefix: "fcn32s/train"
test_initialization: false

```

方框 9.2 solve.py 文件的内容描述

```

import sys
sys.path.append('E:/caffe-windows/python') # 加入 Caffe 的 Python 接口路径
import caffe
import surgery, score
import numpy as np
import os
import setproctitle
setproctitle.setproctitle(os.path.basename(os.getcwd()))
vgg_weights = '../ilsvrc-nets/vgg16-fcn.caffemodel'
vgg_proto = '../ilsvrc-nets/VGG_ILSVRC_16_layers_deploy.prototxt'
caffe.set_device(0)
caffe.set_mode_gpu() # 使用 GPU
solver = caffe.SGDSolver('solver.prototxt') # 使用随机梯度下降求解
vgg_net = caffe.Net(vgg_proto, vgg_weights, caffe.TRAIN) # 加载模型和网络
surgery.transplant(solver.net, vgg_net) # 从 vggnet 中拷贝参数
del vgg_net
interp_layers = [k for k in solver.net.params.keys() if 'up' in k]
# 把全连接层改造成卷积层
surgery.interp(solver.net, interp_layers) # 将卷积核设置成双线性核进行插值
test = np.loadtxt('../data/sift-flow/test.txt', dtype=str) # 导入测试文件目录
for _ in range(50): # 重复 50 次
    solver.step(2000) # 每训练 2000 次进行一次测试
    score.seg_tests(solver, False, test, layer='score_sem', gt='sem')
    # 计算语义分割准确率
    score.seg_tests(solver, False, test, layer='score_geo', gt='geo')
    # 计算几何分割准确率

```

在修改 solver.prototxt 和 solve.py 之后，即可对 SIFT Flow 图像分割案例程序进行训练和测试，其中测试是在训练过程中进行的。程序运行命令如图 9.3 所示，拷贝权值过程如图 9.4 所示，中间结果如图 9.5 所示，最终结果如图 9.6 所示。注意，图 9.4 的拷贝权值过程是为了利用训练好的 VGGNet 来初始化全卷积网络 FCN-32s 的双头版。

从图 9.5 可以看出，在训练 12 980 次时，语义损失为 21 036.6，几何损失为 3985.54。此时，语义测试总准确率约为 82.69%，平均准确率约为 32.72%，平均 IU 约为 25.06%，频权准确率约为 71.78%。几何测试总准确率约为 91.93%，平均准确率约为 93.16%，平均 IU 约为 84.75%，频权准确率约为 85.13%。

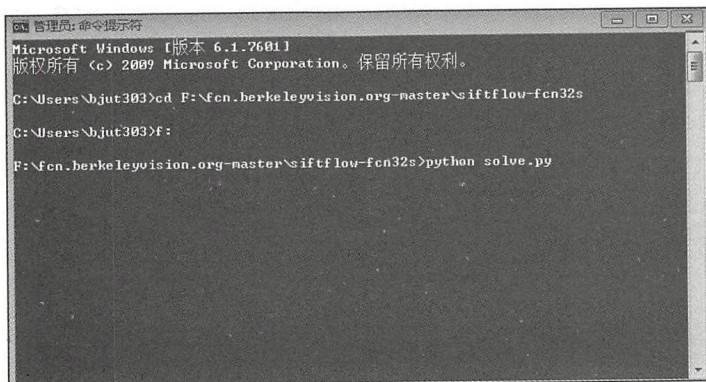


图 9.3 FCN 图像语义和几何分割案例程序的运行命令

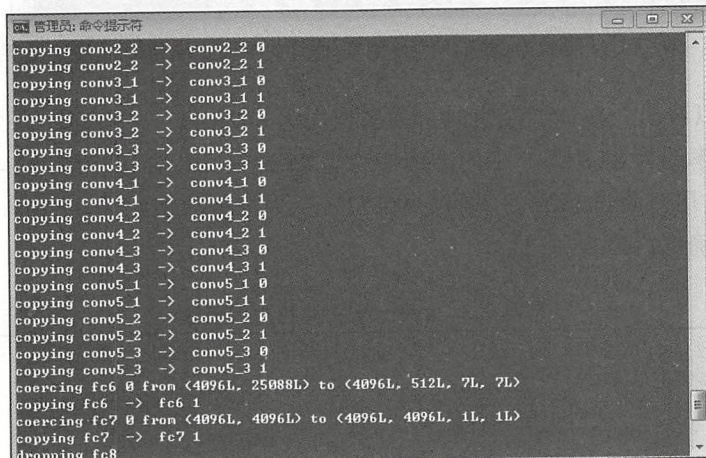


图 9.4 FCN 图像语义和几何分割案例程序的拷贝权值过程

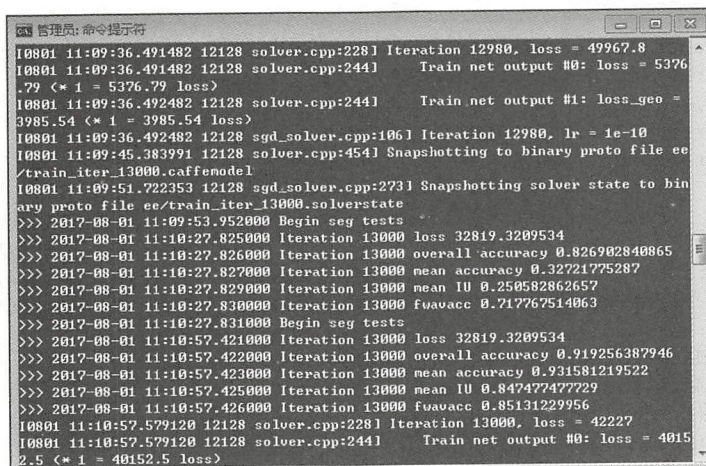


图 9.5 FCN 图像语义和几何分割案例程序的训练中间结果


```

管理员: 命令提示符
I0731 01:31:43.291491 10156 sgd_solver.cpp:1061 Iteration 99960, lr = 1e-10
I0731 01:31:52.653028 10156 solver.cpp:2281 Iteration 99980, loss = 15323.8
I0731 01:31:52.654027 10156 solver.cpp:2441 Train net output #0: loss = 6426
.44 (* 1 = 6426.44 loss)
I0731 01:31:52.654027 10156 solver.cpp:2441 Train net output #1: loss_geo =
2869.41 (* 1 = 2869.41 loss)
I0731 01:31:52.654027 10156 sgd_solver.cpp:1061 Iteration 99980, lr = 1e-10
I0731 01:32:01.546536 10156 solver.cpp:4541 Snapshotting to binary proto file ee
/train_iter_100000.caffemodel
I0731 01:32:07.616883 10156 sgd_solver.cpp:2731 Snapshotting solver state to bin
ary proto file ee/train_iter_100000.solverstate
>>> 2017-07-31 01:32:09.433000 Begin seg tests
>>> 2017-07-31 01:32:43.206000 Iteration 100000 loss 33344.3014285
>>> 2017-07-31 01:32:43.207000 Iteration 100000 overall accuracy 0.84934615424
>>> 2017-07-31 01:32:43.209000 Iteration 100000 mean accuracy 0.452302266966
>>> 2017-07-31 01:32:43.210000 Iteration 100000 mean IU 0.344130407297
>>> 2017-07-31 01:32:43.211000 Iteration 100000 f0avacc 0.754516129266
>>> 2017-07-31 01:32:43.213000 Begin seg tests
>>> 2017-07-31 01:33:12.753000 Iteration 100000 loss 33344.3014285
>>> 2017-07-31 01:33:12.754000 Iteration 100000 overall accuracy 0.937679826332
>>> 2017-07-31 01:33:12.755000 Iteration 100000 mean accuracy 0.937041773293
>>> 2017-07-31 01:33:12.757000 Iteration 100000 mean IU 0.877732202102
>>> 2017-07-31 01:33:12.758000 Iteration 100000 f0avacc 0.883179419765
F:\fcn.berkeleyvision.org-master\siftflow-fcn32s>

```

图 9.6 FCN 图像语义和几何分割案例程序在训练结束时的最终结果

从图 9.6 可以看出，在训练结束时，语义损失为 6426.44，几何损失为 2869.41。此时，语义测试总准确率约为 84.93%，平均准确率约为 45.23%，平均 IU 约为 34.41%，频权准确率约为 75.45%。几何测试总准确率约为 93.77%，平均准确率约为 93.70%，平均 IU 约为 87.77%，频权准确率约为 88.31%。

方框 9.3 infer.py 文件的内容描述

```

import numpy as np
from PIL import Image

import sys
sys.path.append('E:/caffe-windows/python')
import caffe
import matplotlib.pyplot as plt

im = Image.open('coast_arnat59.jpg') # 导入图像
in_ = np.array(im, dtype=np.float32)
in_ = in_[::-1]
in_ = np.array((104.00698793, 116.66876762, 122.67891434))
in_ = in_.transpose((2, 0, 1))

model_def = 'F:/fcn.berkeleyvision.org-master/siftflow-fcn32s/test.prototxt' # 导入网络
model_weights = 'F:/fcn.berkeleyvision.org-master/siftflow-fcn32s/fcn32s/train_
iter_100000.caffemodel' # 导入权值
net = caffe.Net(model_def, model_weights, caffe.TEST)
net.blobs['data'].reshape(1, *in_.shape)
net.blobs['data'].data[...] = in_
net.forward()
plt.subplot(121)
out1 = net.blobs['upscore_sem'].data[0].argmax(axis=0)
plt.imshow(out1) # 展示语义分割结果

```



```
plt.subplot(122)
out2 = net.blobs['upscore_geo'].data[0].argmax(axis=0)
plt.imshow(out2) # 展示几何分割结果
plt.show()
```

最后，在训练完成后，还可以利用方框 9.3，使用如图 9.7 所示的可视化命令，对图像语义分割和几何分割的结果进行可视化。图 9.8 给出了一幅原始图像及其在语义分割和几何分割后的可视化结果。

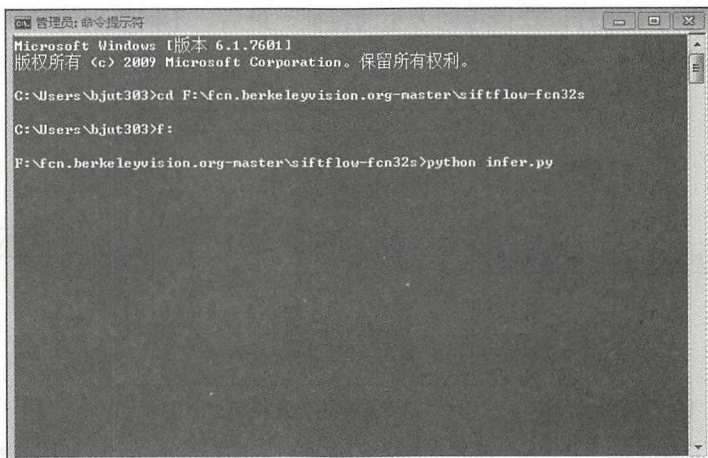


图 9.7 FCN 图像语义和几何分割案例程序的可视化命令

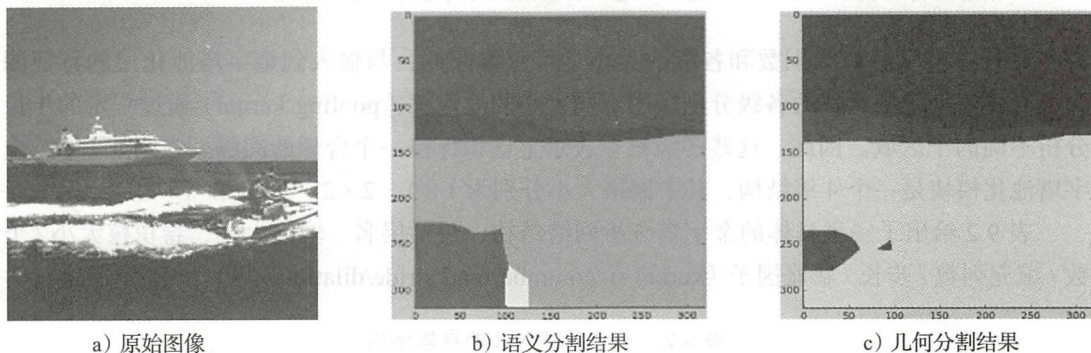


图 9.8 FCN 案例程序的原始图像、语义分割结果和几何分割结果

9.2 金字塔场景分析网络 PSPNet

9.2.1 PSPNet 的模型结构

用全卷积网络进行图像语义分割的主要缺点在于缺少合适的策略来使用全局场景分类线索。金字塔场景分析网络 (Pyramid Scene Parsing Network, PSPNet) 集成了合适的全局

特征进行像素预测^[151]。PSPNet 在 FCN 的基础上，将像素级特征扩展到专门设计的全局金字塔池化特征，通过结合局部和全局线索来提高最终预测的可靠性。

PSPNet 的特点在于使用了金字塔池化模块（pyramid pooling module），其总体结构如图 9.9 所示。金字塔池化模块在 4 个不同的粗细尺度上进行特征融合。最粗尺度对特征图进行全局平均池化，产生单格输出，加细尺度则把特征图分成不同子区域并形成不同位置的池化表示。在金字塔模块中，不同尺度级别的输出包含不同大小的特征图，但都采用 1×1 卷积层把上下文表示的维数降低为原来的 $1/N$ ，其中 N 表示加细级别的大小。然后，低维特征图通过双线性插值进行上采样以获得相同大小的特征。最后，不同级别的特征被拼接为最终的金字塔池化全局特征。

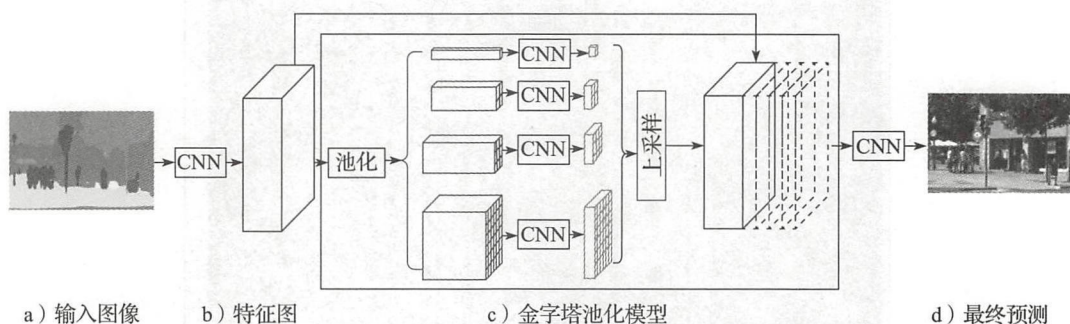


图 9.9 PSPNet 的总体结构。PSPNet 首先把输入图像通过 CNN 产生一个特征图，然后通过金字塔池化模块产生不同的子区域表示，接着通过上采样与特征图的拼接形成包含局部和全局上下文信息的特征表示，最后再经过一个卷积层输出最终的逐像素预测结果

另外，金字塔的级别数和各级的大小是可以修改的，与输入到金字塔池化层的特征图大小有关。金字塔结构的各级分别采用不同大小的池化核（pooling kernel）通过一定的步长分析不同的子区域。因此，这些多级核在表示上应该保持一个合理的间隔。在图 9.9 中，金字塔池化模块是一个 4 级结构，其中网格大小分别为 1×1 、 2×2 、 3×3 和 6×6 。

表 9.2 给出了一个具体的金字塔分析网络结构，包括层名、输出大小、卷积核大小 / 个数 / 填充列数 / 步长 / 膨胀因子（kernel size/number/pad/stride/dilation）等。

表 9.2 一个 PSPNet 的具体结构

部分	layer name	output size	kernel size/number/pad/stride/dilation
1	conv1_1_3×3_s2	237×237	3×3/64/1/2/1
	conv1_2_3×3	237×237	3×3/64/1/1/1
	conv1_3_3×3	237×237	3×3/128/1/1/1
	pool1_3×3_s2	119×119	3×3/64/1/2/1
2	conv2_1_1×1_reduce	119×119	1×1/64/0/1/1
	conv2_1_3×3	119×119	3×3/64/1/1/1
	conv2_1_1×1_increase	119×119	1×1/256/0/1/1

(续)

部分	layer name	output size	kernel size/number/pad/stride/dilation
2	conv2_1_1 × 1_proj	119 × 119	1 × 1/256/0/1/1
	conv2_1	119 × 119	
	conv2_2_1 × 1_reduce	119 × 119	1 × 1/64/0/1/1
	conv2_2_3 × 3	119 × 119	3 × 3/64/1/1/1
	conv2_2_1 × 1_increase	119 × 119	1 × 1/256/0/1/1
	conv2_2	119 × 119	
	conv2_3_1 × 1_reduce	119 × 119	1 × 1/64/0/1/1
	conv2_3_3 × 3	119 × 119	3 × 3/64/1/1/1
	conv2_3_1 × 1_increase	119 × 119	1 × 1/256/0/1/1
	conv2_3	119 × 119	
3	conv3_1_1 × 1_reduce	60 × 60	1 × 1/128/0/2/1
	conv3_1_3 × 3	60 × 60	3 × 3/128/1/1/1
	conv3_1_1 × 1_increase	60 × 60	1 × 1/215/0/1/1
	conv3_1_1 × 1_proj	60 × 60	1 × 1/512/0/2/1
	conv3_1	60 × 60	
	conv3_2_1 × 1_reduce	60 × 60	1 × 1/128/0/1/1
	conv3_2_3 × 3	60 × 60	3 × 3/128/1/1/1
	conv3_2_1 × 1_increase	60 × 60	1 × 1/512/0/1/1
	conv3_2	60 × 60	
	conv3_3_1 × 1_reduce	60 × 60	1 × 1/128/0/1/1
	conv3_3_3 × 3	60 × 60	3 × 3/128/1/1/1
	conv3_3_1 × 1_increase	60 × 60	1 × 1/128/0/1/1
	conv3_3	60 × 60	
	conv3_4_1 × 1_reduce	60 × 60	1 × 1/128/0/1/1
	conv3_4_3 × 3	60 × 60	3 × 3/128/1/1/1
	conv3_4_1 × 1_increase	60 × 60	1 × 1/512/0/1/1
	conv3_4	60 × 60	
4	conv4_1_1 × 1_reduce	60 × 60	1 × 1/256/0/1/1
	conv4_1_3 × 3	60 × 60	3 × 3/256/2/1/2
	conv4_1_1 × 1_increase	60 × 60	1 × 1/1024/0/1/1
	conv4_1_1 × 1_proj	60 × 60	1 × 1/1024/0/1/1
	conv4_1	60 × 60	
	conv4_2_1 × 1_reduce	60 × 60	1 × 1/256/0/1/1
	conv4_2_3 × 3	60 × 60	3 × 3/256/2/1/2
	conv4_2_1 × 1_increase	60 × 60	1 × 1/1024/0/1/1
	conv4_2	60 × 60	
	conv4_3_1 × 1_reduce	60 × 60	1 × 1/256/0/1/1
	conv4_3_3 × 3	60 × 60	3 × 3/256/2/1/2

(续)

部分	layer name	output size	kernel size/number/pad/stride/dilation
4	conv4_3_1×1_increase	60×60	1×1/1024/0/1/1
	conv4_3	60×60	
	conv4_4_1×1_reduce	60×60	1×1/256/0/1/1
	conv4_4_3×3	60×60	3×3/256/2/1/2
	conv4_4_1×1_increase	60×60	1×1/1024/0/1/1
	conv4_4	60×60	
	conv4_5_1×1_reduce	60×60	1×1/256/0/1/1
	conv4_5_3×3	60×60	3×3/256/2/1/2
	conv4_5_1×1_increase	60×60	1×1/1024/0/1/1
	conv4_5	60×60	
	conv4_6_1×1_reduce	60×60	1×1/256/0/1/1
	conv4_6_3×3	60×60	3×3/256/2/1/2
	conv4_6_1×1_increase	60×60	1×1/1024/0/1/1
	conv4_6	60×60	
5	conv5_1_1×1_reduce	60×60	1×1/512/0/1/1
	conv5_1_3×3	60×60	3×3/512/4/1/4
	conv5_1_1×1_increase	60×60	1×1/2048/0/1/1
	conv5_1_1×1_proj	60×60	1×1/2048/0/1/1
	conv5_1	60×60	
	conv5_2_1×1_reduce	60×60	1×1/512/0/1/1
	conv5_2_3×3	60×60	3×3/512/4/1/4
	conv5_2_1×1_increase	60×60	1×1/2048/0/1/1
	conv5_2	60×60	
	conv5_3_1×1_reduce	60×60	1×1/512/0/1/1
	conv5_3_3×3	60×60	3×3/512/4/1/4
	conv5_3_1×1_increase	60×60	1×1/2048/0/1/1
	conv5_3	60×60	
	conv5_3_pool1	1×1	60×60/0/0/60/1
	conv5_3_pool1_conv	1×1	1×1/512/0/1/1
	conv5_3_pool1_interp	60×60	60×60/0/0/0/1
	conv5_3_pool2	2×2	30×30/0/0/30/1
	conv5_3_pool2_conv	2×2	1×1/512/0/1/1
	conv5_3_pool2_interp	60×60	60×60/0/0/0/1
	conv5_3_pool3	3×3	20×20/0/0/20/1
	conv5_3_pool3_conv	3×3	1×1/512/0/1/1
	conv5_3_pool3_interp	60×60	60×60/0/0/0/1
	conv5_3_pool6	6×6	10×10/0/0/10/1
	conv5_3_pool6_conv	6×6	1×1/512/0/1/1

(续)

部分	layer name	output size	kernel size/number/pad/stride/dilation
5	conv5_3_pool6_interp	60 × 60	60 × 60/0/0/0/1
	conv5_3_concat	60 × 60	
	conv5_4	60 × 60	3 × 3/512/1/1/1
6	conv6	60 × 60	1 × 1/150/0/1/1
	conv6_interp	473 × 473	

如表 9.2 所示, 该网络包含 6 个部分, 从 conv1_1_3×3_s2 层到 pool1_3×3_s2 层是第 1 部分, 从 conv2_1_1×1_reduce 层到 conv2_3 层是第 2 部分, 从 conv3_1_1×1_reduce 层到 conv3_4 层是第 3 部分, 从 conv4_1_1×1_reduce 层到 conv4_6 层是第 4 部分, 从 conv5_1_1×1_reduce 层到 conv5_4 层是第 5 部分, 从 conv6 层到 conv6_interp 层是第 6 部分。

在第 1 部分中, conv1_1_3×3_s2 是普通卷积层, 表示第 1 部分的第 1 个残差块中, 使用步长为 2 的 3×3 卷积; pool1_3×3_s2 是池化层, 表示第 1 部分使用步长为 2, 滑动窗口大小为 3×3 的池化。

在第 2 部分中, conv2_1_1×1_reduce、conv2_1_3×3、conv2_1_1×1_increase、conv2_1_1×1_proj、conv2_1_1×1_proj 是普通卷积层, conv2_1 是特征图相加层。其中, conv2_1_1×1_reduce 表示第 2 部分的第 1 个残差块中, 使用 1×1 的卷积进行降维; conv2_1_3×3 表示第 2 部分的第 1 个残差块中, 使用 3×3 卷积核进行卷积; conv2_1_1×1_increase 表示第 2 部分的第 1 个残差块中, 使用 1×1 的卷积进行升维; conv2_1_1×1_proj 表示第 2 部分的第 1 个残差块中, 使用 1×1 的卷积进行匹配维度; conv2_1 表示将该残差块映射后的输入 conv2_1_1×1_proj 与该残差块的输出 conv2_1_1×1_increase 进行相加, 相加后的结果作为相邻下一残差块的输入。

第 3 部分和第 4 部分的结构与第 2 部分类似。需要注意的是, 第 4 部分除了使用普通卷积层之外, 还使用了膨胀卷积层。例如, conv4_1_3×3 的膨胀因子 = 2, 所以是膨胀卷积层, 表示第 4 部分第 1 个残差块使用膨胀因子为 2、卷积核大小为 3×3 进行膨胀卷积。

第 5 部分包含 3 个残差块和金字塔模块。其中, conv5_1_1×1_reduce 层至 conv5_1 层是第 1 个残差块, conv5_2_1×1_reduce 层至 conv5_2 层是第 2 个残差块, conv5_3_1×1_reduce 层至 conv5_3 层是第 3 个残差块。这两个残差块与第 4 部分的残差块类似, 即残差块中不仅包含普通卷积层, 而且包含膨胀卷积层。conv5_3_pool1 层至 conv5_3_concat 层是金字塔模块。其中, conv5_3_pool1, conv5_3_pool1_conv, conv5_3_pool1_interp 表示第 1 个层级的一系列操作。conv5_3_pool1 表示对特征图进行全局池化, 得到网格大小为 1×1 的池化结果; conv5_3_pool1_conv 表示对池化结果进行卷积; conv5_3_pool1_interp 表示对卷积后的池化结果进行插值, 得到目标大小的特征图。分别经过 4 种层级的操作之后, 再由 conv5_3_concat 层进行拼接, 得到整个金字塔模块的结果。

第 6 部分包含 conv6 层和 conv6_interp 层。其中, conv6 层是普通卷积层, 表示对第 5

部分得到的结果进行卷积；conv6_interp 层是插值层，表示对 conv6 层的卷积结果进行插值，得到与输入大小相同的输出。

根据上述分析不难发现，输入图像对应图 9.9a、conv5_3 层对应图 9.9b、conv5_3_pool1 层至 conv5_3_concat 层是对应图 9.9c 的金字塔模块，conv6_interp 层是对应图 9.9d 的最终结果。

9.2.2 PSPNet 的 TensorFlow 代码实现及说明

关于 PSPNet 的 TensorFlow 代码，下载地址为 <https://github.com/hellochick/PSPNet-tensorflow>。这个网址的文件夹里有多文件子文件夹，例如训练模型文件 train.py、模型定义文件 model.py 和可视化文件 inference.py 等，以及存放训练集和验证集列表文件夹 list、输入可视化图像文件夹 input 和输出可视化图像文件夹 output 等。考虑到有些代码与之前的模型有重复，下面仅对训练模型文件 train.py 和模型定义文件 model.py 分别进行详细介绍和说明。

1. 训练模型文件 train.py 代码及说明

```
from __future__ import print_function

import argparse
import os
import sys
import time
import tensorflow as tf
import numpy as np
from model import PSPNet101
from tools import prepare_label
from image_reader import ImageReader

IMG_MEAN = np.array((103.939, 116.779, 123.68), dtype=np.float32) # 图像均值

BATCH_SIZE = 1
DATA_DIRECTORY = './'
DATA_LIST_PATH = './list/ade20k_train_list.txt'
IGNORE_LABEL = 255
INPUT_SIZE = '720,720'
LEARNING_RATE = 1e-3
MOMENTUM = 0.9
NUM_CLASSES = 150
NUM_STEPS = 60001
POWER = 0.9
RANDOM_SEED = 1234
WEIGHT_DECAY = 0.0001
RESTORE_FROM = './model'
SNAPSHOT_DIR = './model/ade20k_model/pspnet50/'
SAVE_NUM_IMAGES = 4
SAVE_PRED_EVERY = 1000
```


GPU_MEMORY_FRACTION=0.9

```
def get_arguments():
    # 获得命令行参数
    parser = argparse.ArgumentParser(description="DeepLab-ResNet Network")
    parser.add_argument("--batch-size", type=int, default=BATCH_SIZE,
                        help="Number of images sent to the network in one step.")
    parser.add_argument("--data-dir", type=str, default=DATA_DIRECTORY,
                        help="Path to the directory containing the PASCAL VOC dataset.")
    parser.add_argument("--data-list", type=str, default=DATA_LIST_PATH,
                        help="Path to the file listing the images in the dataset.")
    parser.add_argument("--ignore-label", type=int, default=IGNORE_LABEL,
                        help="The index of the label to ignore during the training.")
    parser.add_argument("--input-size", type=str, default=INPUT_SIZE,
                        help="Comma-separated string with height and width of images.")
    parser.add_argument("--is-training", action="store_true",
                        help="Whether to updates the running means and variances during the training.")
    parser.add_argument("--learning-rate", type=float, default=LEARNING_RATE,
                        help="Base learning rate for training with polynomial decay.")
    parser.add_argument("--momentum", type=float, default=MOMENTUM,
                        help="Momentum component of the optimiser.")
    parser.add_argument("--not-restore-last", action="store_true",
                        help="Whether to not restore last (FC) layers.")
    parser.add_argument("--num-classes", type=int, default=NUM_CLASSES,
                        help="Number of classes to predict (including background).")
    parser.add_argument("--num-steps", type=int, default=NUM_STEPS,
                        help="Number of training steps.")
    parser.add_argument("--power", type=float, default=POWER,
                        help="Decay parameter to compute the learning rate.")
    parser.add_argument("--random-mirror", action="store_true",
                        help="Whether to randomly mirror the inputs during the training.")
    parser.add_argument("--random-scale", action="store_true",
                        help="Whether to randomly scale the inputs during the training.")
    parser.add_argument("--random-seed", type=int, default=RANDOM_SEED,
                        help="Random seed to have reproducible results.")
    parser.add_argument("--restore-from", type=str, default=RESTORE_FROM,
                        help="Where restore model parameters from.")
    parser.add_argument("--save-num-images", type=int, default=SAVE_NUM_IMAGES,
                        help="How many images to save.")
    parser.add_argument("--save-pred-every", type=int, default=SAVE_PRED_EVERY,
                        help="Save summaries and checkpoint every often.")
    parser.add_argument("--snapshot-dir", type=str, default=SNAPSHOT_DIR,
                        help="Where to save snapshots of the model.")
    parser.add_argument("--weight-decay", type=float, default=WEIGHT_DECAY,
                        help="Regularisation parameter for L2-loss.")
    parser.add_argument("--update-mean-var", action="store_true",
                        help="whether to get update_op from tf.Graphic_Keys")
    parser.add_argument("--train-beta-gamma", action="store_true",
```

```

        help="whether to train beta & gamma in bn layer")
    return parser.parse_args()

def save(saver, sess, logdir, step):          # 设置保存的权值文件名和存放路径
    model_name = 'model.ckpt'
    checkpoint_path = os.path.join(logdir, model_name)

    if not os.path.exists(logdir):           # 保存权值文件
        os.makedirs(logdir)
    saver.save(sess, checkpoint_path, global_step=step)
    print('The checkpoint has been created.')

def load(saver, sess, ckpt_path):            # 导入权值文件
    saver.restore(sess, ckpt_path)
    print("Restored model parameters from {}".format(ckpt_path))

def main():                                  # 创建模型，开始训练
    args = get_arguments()

    h, w = map(int, args.input_size.split(','))
    input_size = (h, w)
    tf.set_random_seed(args.random_seed)
    coord = tf.train.Coordinator()

    with tf.name_scope("create_inputs"):     # 创建输入
        reader = ImageReader(args.data_dir, args.data_list, input_size, args.\
            random_scale, args.random_mirror, args.ignore_label, IMG_MEAN, coord)
        image_batch, label_batch = reader.dequeue(args.batch_size)

    net = PSPNet50({'data': image_batch}, is_training=True, num_classes=args.\
        num_classes)
    raw_output = net.layers['conv6']

    fc_list = ['conv5_3_pool1_conv', 'conv5_3_pool2_conv', 'conv5_3_pool3_conv',
        'conv5_3_pool6_conv', 'conv6', 'conv5_4']
    restore_var = [v for v in tf.global_variables()]
    all_trainable = [v for v in tf.trainable_variables() if ('beta' not in
        v.name and 'gamma' not in v.name) or args.train_beta_gamma]
    fc_trainable = [v for v in all_trainable if v.name.split('/')[0] in fc_list]
    conv_trainable = [v for v in all_trainable if v.name.split('/')[0] not in\
        fc_list] # lr * 1.0
    fc_w_trainable = [v for v in fc_trainable if 'weights' in v.name] # lr * 10.0
    fc_b_trainable = [v for v in fc_trainable if 'biases' in v.name] # lr * 20.0
    assert(len(all_trainable) == len(fc_trainable) + len(conv_trainable))
    assert(len(fc_trainable) == len(fc_w_trainable) + len(fc_b_trainable))

    # 预测
    raw_prediction = tf.reshape(raw_output, [-1, args.num_classes])
    label_proc = prepare_label(label_batch, tf.stack(raw_output.get_shape())\
        [1:3]), num_classes=args.num_classes, one_hot=False) # [batch_size, h, w]
    raw_gt = tf.reshape(label_proc, [-1,])
    indices = tf.squeeze(tf.where(tf.less_equal(raw_gt, args.num_classes - 1)), 1)
    gt = tf.cast(tf.gather(raw_gt, indices), tf.int32)

```

```

prediction = tf.gather(raw_prediction, indices)

# softmax 损失
loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=prediction, labels=gt)
l2_losses = [args.weight_decay * tf.nn.l2_loss(v) for v in tf.trainable_variables()
              if 'weights' in v.name]
reduced_loss = tf.reduce_mean(loss) + tf.add_n(l2_losses)

# 使用 Poly 学习率策略
base_lr = tf.constant(args.learning_rate)
step_ph = tf.placeholder(dtype=tf.float32, shape=())
learning_rate = tf.scalar_mul(base_lr, tf.pow((1 - step_ph / args.num_steps),
                                              args.power))

# 更新移动平均和移动方差
if args.update_mean_var == False:
    update_ops = None
else:
    update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)

with tf.control_dependencies(update_ops):
    opt_conv = tf.train.MomentumOptimizer(learning_rate, args.momentum)
    opt_fc_w = tf.train.MomentumOptimizer(learning_rate * 10.0, args.momentum)
    opt_fc_b = tf.train.MomentumOptimizer(learning_rate * 20.0, args.momentum)

    grads = tf.gradients(reduced_loss, conv_trainable + fc_w_trainable + \
                          fc_b_trainable)
    grads_conv = grads[:len(conv_trainable)]
    grads_fc_w = grads[len(conv_trainable) : (len(conv_trainable) + \
                                              len(fc_w_trainable))]
    grads_fc_b = grads[(len(conv_trainable) + len(fc_w_trainable)):]

    train_op_conv = opt_conv.apply_gradients(zip(grads_conv, conv_trainable))
    train_op_fc_w = opt_fc_w.apply_gradients(zip(grads_fc_w, fc_w_trainable))
    train_op_fc_b = opt_fc_b.apply_gradients(zip(grads_fc_b, fc_b_trainable))
    train_op = tf.group(train_op_conv, train_op_fc_w, train_op_fc_b)

# 建立 tf 会话过程，并初始化权值变量
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session(config=config)
init = tf.global_variables_initializer()
sess.run(init)

saver = tf.train.Saver(var_list=tf.global_variables(), max_to_keep=10)
# 保存训练权值

ckpt = tf.train.get_checkpoint_state(SNAPSHOT_DIR)
if ckpt and ckpt.model_checkpoint_path:
    loader = tf.train.Saver(var_list=restore_var)
    load_step = int(os.path.basename(ckpt.model_checkpoint_path).split('-')[1])
    load(loader, sess, ckpt.model_checkpoint_path)
else:

```



```

print('No checkpoint file found.')
load_step = 0

threads = tf.train.start_queue_runners(coord=coord, sess=sess) # 启动队列线程

for step in range(args.num_steps): # 在训练中进行迭代
    start_time = time.time()
    feed_dict = {step_ph: step}
    if step % args.save_pred_every == 0:
        loss_value, _ = sess.run([reduced_loss, train_op], feed_dict=feed_dict)
        save(saver, sess, args.snapshot_dir, step)
    else:
        loss_value, _ = sess.run([reduced_loss, train_op], feed_dict=feed_dict)
    duration = time.time() - start_time
    print('step {:d} \t loss = {:.3f}, ({:.3f} sec/step)'.format(step, \
        loss_value, duration))

coord.request_stop()
coord.join(threads)

if __name__ == '__main__':
    main()

```

2. 模型定义文件 model.py 代码及说明

该文件使用 PSPNet101 和 PSPNet50 两个类，定义了 PSPNet 的两种网络结构。这里使用 PSPNet50 的网络进行图像语义分割，因此下面只展示 PSPNet50 的代码并进行说明。有兴趣的读者可查看 model.py 文件了解 PSPNet101 的网络结构。

```

from network import Network
import tensorflow as tf

class PSPNet50(Network): # 模型定义
    def setup(self, is_training, num_classes):
        (self.feed('data') # 导入数据
         .conv(3, 3, 64, 2, 2, biased=False, relu=False, padding='SAME',
              name='conv1_1_3x3_s2')
         .batch_normalization(relu=False, name='conv1_1_3x3_s2_bn')
         .relu(name='conv1_1_3x3_s2_bn_relu')
         .conv(3, 3, 64, 1, 1, biased=False, relu=False, padding='SAME',
              name='conv1_2_3x3')
         .batch_normalization(relu=True, name='conv1_2_3x3_bn')
         .conv(3, 3, 128, 1, 1, biased=False, relu=False, padding='SAME',
              name='conv1_3_3x3')
         .batch_normalization(relu=True, name='conv1_3_3x3_bn')
         .max_pool(3, 3, 2, 2, padding='SAME', name='pool1_3x3_s2')
         .conv(1, 1, 256, 1, 1, biased=False, relu=False, name='conv2_1_1x1_\
proj')
         .batch_normalization(relu=False, name='conv2_1_1x1_proj_bn'))

        (self.feed('pool1_3x3_s2') # 导入 pool1_3x3_s2 层

```

```

        .conv(1, 1, 64, 1, 1, biased=False, relu=False, name='conv2_1_1x1_\
            reduce')
        .batch_normalization(relu=True, name='conv2_1_1x1_reduce_bn')
        .zero_padding(paddings=1, name='padding1')
        .conv(3, 3, 64, 1, 1, biased=False, relu=False, name='conv2_1_3x3')
        .batch_normalization(relu=True, name='conv2_1_3x3_bn')
        .conv(1, 1, 256, 1, 1, biased=False, relu=False, name='conv2_1_1x1_\
            increase')
        .batch_normalization(relu=False, name='conv2_1_1x1_increase_bn'))

    (self.feed('conv2_1_1x1_proj_bn', 'conv2_1_1x1_increase_bn')
     .add(name='conv2_1')
     .relu(name='conv2_1/relu')
     .conv(1, 1, 64, 1, 1, biased=False, relu=False, name='conv2_2_1x1_\
         reduce')
     .batch_normalization(relu=True, name='conv2_2_1x1_reduce_bn')
     .zero_padding(paddings=1, name='padding2')
     .conv(3, 3, 64, 1, 1, biased=False, relu=False, name='conv2_2_3x3')
     .batch_normalization(relu=True, name='conv2_2_3x3_bn')
     .conv(1, 1, 256, 1, 1, biased=False, relu=False, name='conv2_2_1x1_\
         increase')
     .batch_normalization(relu=False, name='conv2_2_1x1_increase_bn'))

    (self.feed('conv2_1/relu', 'conv2_2_1x1_increase_bn')
     .add(name='conv2_2')
     .relu(name='conv2_2/relu')
     .conv(1, 1, 64, 1, 1, biased=False, relu=False, name='conv2_3_1x1_\
         reduce')
     .batch_normalization(relu=True, name='conv2_3_1x1_reduce_bn')
     .zero_padding(paddings=1, name='padding3')
     .conv(3, 3, 64, 1, 1, biased=False, relu=False, name='conv2_3_3x3')
     .batch_normalization(relu=True, name='conv2_3_3x3_bn')
     .conv(1, 1, 256, 1, 1, biased=False, relu=False, name='conv2_3_1x1_\
         increase')
     .batch_normalization(relu=False, name='conv2_3_1x1_increase_bn'))

    (self.feed('conv2_2/relu', 'conv2_3_1x1_increase_bn')
     .add(name='conv2_3')
     .relu(name='conv2_3/relu')
     .conv(1, 1, 512, 2, 2, biased=False, relu=False, name='conv3_1_1x1_\
         proj')
     .batch_normalization(relu=False, name='conv3_1_1x1_proj_bn'))

    (self.feed('conv2_3/relu')
     # 导入 conv2_3/ReLU 层
     .conv(1, 1, 128, 2, 2, biased=False, relu=False, name='conv3_1_1x1_\
         reduce')
     .batch_normalization(relu=True, name='conv3_1_1x1_reduce_bn')
     .zero_padding(paddings=1, name='padding4')
     .conv(3, 3, 128, 1, 1, biased=False, relu=False, name='conv3_1_3x3')
     .batch_normalization(relu=True, name='conv3_1_3x3_bn')
     .conv(1, 1, 512, 1, 1, biased=False, relu=False, name='conv3_1_1x1_\
         increase')
     .batch_normalization(relu=False, name='conv3_1_1x1_increase_bn'))

```

```

(self.feed('conv3_1_1x1_proj_bn', 'conv3_1_1x1_increase_bn')
 .add(name='conv3_1')
 .relu(name='conv3_1/relu')
 .conv(1, 1, 128, 1, 1, biased=False, relu=False, name='conv3_2_1x1_\
      reduce')
 .batch_normalization(relu=True, name='conv3_2_1x1_reduce_bn')
 .zero_padding(paddings=1, name='padding5')
 .conv(3, 3, 128, 1, 1, biased=False, relu=False, name='conv3_2_3x3')
 .batch_normalization(relu=True, name='conv3_2_3x3_bn')
 .conv(1, 1, 512, 1, 1, biased=False, relu=False, name='conv3_2_1x1_\
      increase')
 .batch_normalization(relu=False, name='conv3_2_1x1_increase_bn'))

(self.feed('conv3_1/relu', 'conv3_2_1x1_increase_bn')
 .add(name='conv3_2')
 .relu(name='conv3_2/relu')
 .conv(1, 1, 128, 1, 1, biased=False, relu=False, name='conv3_3_1x1_\
      reduce')
 .batch_normalization(relu=True, name='conv3_3_1x1_reduce_bn')
 .zero_padding(paddings=1, name='padding6')
 .conv(3, 3, 128, 1, 1, biased=False, relu=False, name='conv3_3_3x3')
 .batch_normalization(relu=True, name='conv3_3_3x3_bn')
 .conv(1, 1, 512, 1, 1, biased=False, relu=False, name='conv3_3_1x1_\
      increase')
 .batch_normalization(relu=False, name='conv3_3_1x1_increase_bn'))

(self.feed('conv3_2/relu', 'conv3_3_1x1_increase_bn')
 .add(name='conv3_3')
 .relu(name='conv3_3/relu')
 .conv(1, 1, 128, 1, 1, biased=False, relu=False, name='conv3_4_1x1_\
      reduce')
 .batch_normalization(relu=True, name='conv3_4_1x1_reduce_bn')
 .zero_padding(paddings=1, name='padding7')
 .conv(3, 3, 128, 1, 1, biased=False, relu=False, name='conv3_4_3x3')
 .batch_normalization(relu=True, name='conv3_4_3x3_bn')
 .conv(1, 1, 512, 1, 1, biased=False, relu=False, name='conv3_4_1x1_\
      increase')
 .batch_normalization(relu=False, name='conv3_4_1x1_increase_bn'))

(self.feed('conv3_3/relu', 'conv3_4_1x1_increase_bn')
 .add(name='conv3_4')
 .relu(name='conv3_4/relu')
 .conv(1, 1, 1024, 1, 1, biased=False, relu=False, name='conv4_1_1x1_\
      proj')
 .batch_normalization(relu=False, name='conv4_1_1x1_proj_bn'))

(self.feed('conv3_4/relu')
 .conv(1, 1, 256, 1, 1, biased=False, relu=False, name='conv4_1_1x1_\
      reduce')
 .batch_normalization(relu=True, name='conv4_1_1x1_reduce_bn')
 .zero_padding(paddings=2, name='padding8')
 .atrous_conv(3, 3, 256, 2, biased=False, relu=False, name='conv4_1_3x3')
 .batch_normalization(relu=True, name='conv4_1_3x3_bn'))

```



```

        .conv(1, 1, 1024, 1, 1, biased=False, relu=False, name='conv4_1_1x1_\
            increase')
        .batch_normalization(relu=False, name='conv4_1_1x1_increase_bn'))

    (self.feed('conv4_1_1x1_proj_bn', 'conv4_1_1x1_increase_bn')
     .add(name='conv4_1')
     .relu(name='conv4_1/relu')
     .conv(1, 1, 256, 1, 1, biased=False, relu=False, name='conv4_2_1x1_\
         reduce')
     .batch_normalization(relu=True, name='conv4_2_1x1_reduce_bn')
     .zero_padding(paddings=2, name='padding9')
     .atrous_conv(3, 3, 256, 2, biased=False, relu=False, name='conv4_2_3x3')
     .batch_normalization(relu=True, name='conv4_2_3x3_bn')
     .conv(1, 1, 1024, 1, 1, biased=False, relu=False, name='conv4_2_1x1_\
         increase')
     .batch_normalization(relu=False, name='conv4_2_1x1_increase_bn'))

    (self.feed('conv4_1/relu', 'conv4_2_1x1_increase_bn')
     .add(name='conv4_2')
     .relu(name='conv4_2/relu')
     .conv(1, 1, 256, 1, 1, biased=False, relu=False, name='conv4_3_1x1_\
         reduce')
     .batch_normalization(relu=True, name='conv4_3_1x1_reduce_bn')
     .zero_padding(paddings=2, name='padding10')
     .atrous_conv(3, 3, 256, 2, biased=False, relu=False, name='conv4_3_3x3')
     .batch_normalization(relu=True, name='conv4_3_3x3_bn')
     .conv(1, 1, 1024, 1, 1, biased=False, relu=False, name='conv4_3_1x1_\
         increase')
     .batch_normalization(relu=False, name='conv4_3_1x1_increase_bn'))

    (self.feed('conv4_2/relu', 'conv4_3_1x1_increase_bn')
     .add(name='conv4_3')
     .relu(name='conv4_3/relu')
     .conv(1, 1, 256, 1, 1, biased=False, relu=False, name='conv4_4_1x1_\
         reduce')
     .batch_normalization(relu=True, name='conv4_4_1x1_reduce_bn')
     .zero_padding(paddings=2, name='padding11')
     .atrous_conv(3, 3, 256, 2, biased=False, relu=False, name='conv4_4_3x3')
     .batch_normalization(relu=True, name='conv4_4_3x3_bn')
     .conv(1, 1, 1024, 1, 1, biased=False, relu=False, name='conv4_4_1x1_\
         increase')
     .batch_normalization(relu=False, name='conv4_4_1x1_increase_bn'))

    (self.feed('conv4_3/relu', 'conv4_4_1x1_increase_bn')
     .add(name='conv4_4')
     .relu(name='conv4_4/relu')
     .conv(1, 1, 256, 1, 1, biased=False, relu=False, name='conv4_5_1x1_\
         reduce')
     .batch_normalization(relu=True, name='conv4_5_1x1_reduce_bn')
     .zero_padding(paddings=2, name='padding12')
     .atrous_conv(3, 3, 256, 2, biased=False, relu=False, name='conv4_5_3x3')
     .batch_normalization(relu=True, name='conv4_5_3x3_bn')
     .conv(1, 1, 1024, 1, 1, biased=False, relu=False, name='conv4_5_1x1_\

```

```

        increase'))
    .batch_normalization(relu=False, name='conv4_5_1x1_increase_bn'))

    (self.feed('conv4_4/relu', 'conv4_5_1x1_increase_bn'))
    .add(name='conv4_5')
    .relu(name='conv4_5/relu')
    .conv(1, 1, 256, 1, 1, biased=False, relu=False, name='conv4_6_1x1_\
        reduce')
    .batch_normalization(relu=True, name='conv4_6_1x1_reduce_bn')
    .zero_padding(paddings=2, name='padding13')
    .atrous_conv(3, 3, 256, 2, biased=False, relu=False, name='conv4_6_3x3')
    .batch_normalization(relu=True, name='conv4_6_3x3_bn')
    .conv(1, 1, 1024, 1, 1, biased=False, relu=False, name='conv4_6_1x1_\
        increase')
    .batch_normalization(relu=False, name='conv4_6_1x1_increase_bn'))

    (self.feed('conv4_5/relu', 'conv4_6_1x1_increase_bn'))
    .add(name='conv4_6')
    .relu(name='conv4_6/relu')
    .conv(1, 1, 2048, 1, 1, biased=False, relu=False, name='conv5_1_1x1_\
        proj')
    .batch_normalization(relu=False, name='conv5_1_1x1_proj_bn'))

    (self.feed('conv4_6/relu')                                     # 导入 conv4_6/ReLU 层
    .conv(1, 1, 512, 1, 1, biased=False, relu=False,
        name='conv5_1_1x1_reduce')
    .batch_normalization(relu=True, name='conv5_1_1x1_reduce_bn')
    .zero_padding(paddings=4, name='padding31')
    .atrous_conv(3, 3, 512, 4, biased=False, relu=False, name='conv5_1_3x3')
    .batch_normalization(relu=True, name='conv5_1_3x3_bn')
    .conv(1, 1, 2048, 1, 1, biased=False, relu=False, name='conv5_1_\
        1x1_increase')
    .batch_normalization(relu=False, name='conv5_1_1x1_increase_bn'))

    (self.feed('conv5_1_1x1_proj_bn', 'conv5_1_1x1_increase_bn'))
    .add(name='conv5_1')
    .relu(name='conv5_1/relu')
    .conv(1, 1, 512, 1, 1, biased=False, relu=False, name='conv5_2_1x1_\
        reduce')
    .batch_normalization(relu=True, name='conv5_2_1x1_reduce_bn')
    .zero_padding(paddings=4, name='padding32')
    .atrous_conv(3, 3, 512, 4, biased=False, relu=False, name='conv5_2_3x3')
    .batch_normalization(relu=True, name='conv5_2_3x3_bn')
    .conv(1, 1, 2048, 1, 1, biased=False, relu=False, name='conv5_2_1x1_\
        increase')
    .batch_normalization(relu=False, name='conv5_2_1x1_increase_bn'))

    (self.feed('conv5_1/relu', 'conv5_2_1x1_increase_bn'))
    .add(name='conv5_2')
    .relu(name='conv5_2/relu')
    .conv(1, 1, 512, 1, 1, biased=False, relu=False, name='conv5_3_1x1_\
        reduce')
    .batch_normalization(relu=True, name='conv5_3_1x1_reduce_bn'))

```

```

.zero_padding(paddings=4, name='padding33')
.atrous_conv(3, 3, 512, 4, biased=False, relu=False, name='conv5_3_3x3')
.batch_normalization(relu=True, name='conv5_3_3x3_bn')
.conv(1, 1, 2048, 1, 1, biased=False, relu=False, name='conv5_3_1x1_\
    increase')
.batch_normalization(relu=False, name='conv5_3_1x1_increase_bn'))

(self.feed('conv5_2/relu', 'conv5_3_1x1_increase_bn')
.add(name='conv5_3')
.relu(name='conv5_3/relu'))

conv5_3 = self.layers['conv5_3/relu']
shape = tf.shape(conv5_3)[1:3]

(self.feed('conv5_3/relu')                                # 导入 conv5_3/ReLU 层
.avg_pool(60, 60, 60, 60, name='conv5_3_pool11')
.conv(1, 1, 512, 1, 1, biased=False, relu=False, name='conv5_3_pool11_\
    conv')
.batch_normalization(relu=True, name='conv5_3_pool11_conv_bn')
.resize_bilinear(shape, name='conv5_3_pool11_interp'))

(self.feed('conv5_3/relu')                                # 导入 conv5_3/ReLU 层
.avg_pool(30, 30, 30, 30, name='conv5_3_pool12')
.conv(1, 1, 512, 1, 1, biased=False, relu=False, name='conv5_3_pool12_\
    conv')
.batch_normalization(relu=True, name='conv5_3_pool12_conv_bn')
.resize_bilinear(shape, name='conv5_3_pool12_interp'))

(self.feed('conv5_3/relu')                                # 导入 conv5_3/ReLU 层
.avg_pool(20, 20, 20, 20, name='conv5_3_pool13')
.conv(1, 1, 512, 1, 1, biased=False, relu=False, name='conv5_3_pool13_\
    conv')
.batch_normalization(relu=True, name='conv5_3_pool13_conv_bn')
.resize_bilinear(shape, name='conv5_3_pool13_interp'))

(self.feed('conv5_3/relu')                                # 导入 conv5_3/ReLU 层
.avg_pool(10, 10, 10, 10, name='conv5_3_pool16')
.conv(1, 1, 512, 1, 1, biased=False, relu=False, name='conv5_3_pool16_\
    conv')
.batch_normalization(relu=True, name='conv5_3_pool16_conv_bn')
.resize_bilinear(shape, name='conv5_3_pool16_interp'))

(self.feed('conv5_3/relu', 'conv5_3_pool16_interp', 'conv5_3_pool13_interp',
    'conv5_3_pool12_interp', 'conv5_3_pool11_interp')
.concat(axis=-1, name='conv5_3_concat')
.conv(3, 3, 512, 1, 1, biased=False, relu=False, padding='SAME', name='
    conv5_4')
.batch_normalization(relu=True, name='conv5_4_bn')
.conv(1, 1, num_classes, 1, 1, biased=True, relu=False, name='conv6'))

```

9.2.3 PSPNet 的图像语义分割案例及演示效果

本节描述一个利用 PSPNet 在 TensorFlow 框架下进行图像实例分割的案例，其中用到

的 ADE20K 数据集可以根据表 1.2 提供的地址下载。还需要下载初始化权值文件 model.ckpt 并存放到 /PSPNet-tensorflow-master/model/ 文件夹下。

在一切准备就绪后，即可参照图 9.10 的训练命令运行 PSPNet 的图像语义分割案例程序，图 9.11 给出了训练中间信息。图 9.12 给出了网络在训练结束时的显示信息。在训练结束之后，即可参照图 9.13 的命令验证图像语义分割的效果，如图 9.14 所示。注意，图 9.13 中的 CUDA_VISIBLE_DEVICES=1 表示使用编号为 1 的 GPU。

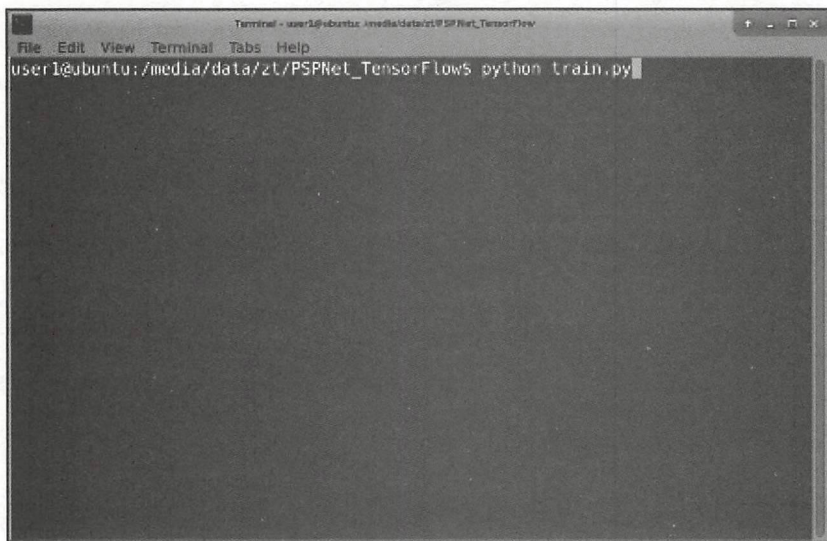


图 9.10 PSPNet 图像语义分割案例的训练命令

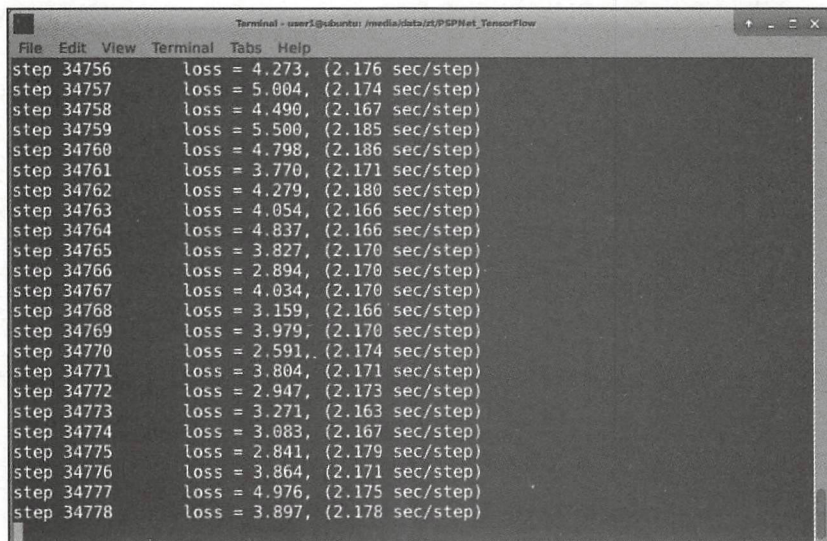
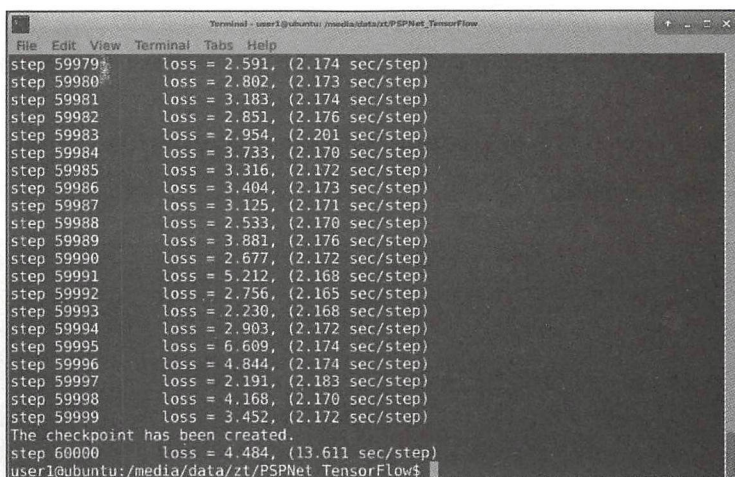


图 9.11 PSPNet 图像语义分割案例的训练信息



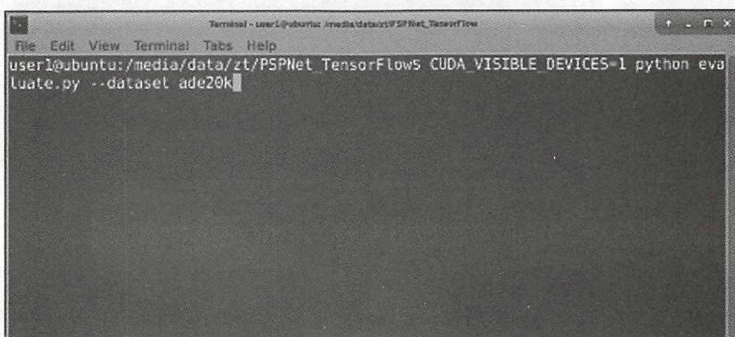


```

File Edit View Terminal Tabs Help
step 59979 loss = 2.591, (2.174 sec/step)
step 59980 loss = 2.802, (2.173 sec/step)
step 59981 loss = 3.183, (2.174 sec/step)
step 59982 loss = 2.851, (2.176 sec/step)
step 59983 loss = 2.954, (2.201 sec/step)
step 59984 loss = 3.733, (2.170 sec/step)
step 59985 loss = 3.316, (2.172 sec/step)
step 59986 loss = 3.404, (2.173 sec/step)
step 59987 loss = 3.125, (2.171 sec/step)
step 59988 loss = 2.533, (2.170 sec/step)
step 59989 loss = 3.881, (2.176 sec/step)
step 59990 loss = 2.677, (2.172 sec/step)
step 59991 loss = 5.212, (2.168 sec/step)
step 59992 loss = 2.756, (2.165 sec/step)
step 59993 loss = 2.230, (2.168 sec/step)
step 59994 loss = 2.903, (2.172 sec/step)
step 59995 loss = 6.609, (2.174 sec/step)
step 59996 loss = 4.844, (2.174 sec/step)
step 59997 loss = 2.191, (2.183 sec/step)
step 59998 loss = 4.168, (2.170 sec/step)
step 59999 loss = 3.452, (2.172 sec/step)
The checkpoint has been created.
step 60000 loss = 4.484, (13.611 sec/step)
user1@ubuntu:/media/data/zt/PSPNet_TensorFlow$

```

图 9.12 PSPNet 图像语义分割案例程序在训练结束时的显示信息

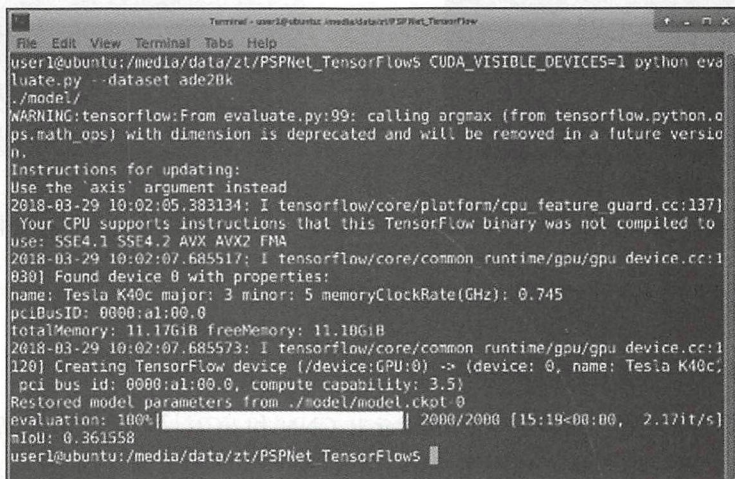


```

File Edit View Terminal Tabs Help
user1@ubuntu:/media/data/zt/PSPNet_TensorFlow$ CUDA_VISIBLE_DEVICES=1 python evaluate.py --dataset ade20k

```

图 9.13 PSPNet 图像语义分割案例程序的验证命令



```

File Edit View Terminal Tabs Help
user1@ubuntu:/media/data/zt/PSPNet_TensorFlow$ CUDA_VISIBLE_DEVICES=1 python evaluate.py --dataset ade20k
./model/
WARNING:tensorflow:From evaluate.py:99: calling argmax (from tensorflow.python.ops.math_ops) with dimension is deprecated and will be removed in a future version.
Instructions for updating:
Use the 'axis' argument instead
2018-03-29 10:02:05.383134: I tensorflow/core/platform/cpu_feature_guard.cc:137] Your CPU supports instructions that this TensorFlow binary was not compiled to use: SSE4.1 SSE4.2 AVX AVX2 FMA
2018-03-29 10:02:07.685517: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1030] Found device 0 with properties:
name: Tesla K40c major: 3 minor: 5 memoryClockRate(GHz): 0.745
pciBusID: 0000:a1:00:0
totalMemory: 11.17GiB freeMemory: 11.10GiB
2018-03-29 10:02:07.685573: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1120] Creating TensorFlow device (/device:GPU:0) -> (device: 0, name: Tesla K40c, pci bus id: 0000:a1:00:0, compute capability: 3.5)
Restored model parameters from ./model/model.ckpt-0
evaluation: 100% [████████████████████████████████████████] 2000/2000 [15:19<00:00, 2.17it/s]
mIoU: 0.361558
user1@ubuntu:/media/data/zt/PSPNet_TensorFlow$

```

图 9.14 PSPNet 图像语义分割案例程序的验证结果



从图 9.11 可以看出，在训练 34 778 次时，语义损失为 3.897。从图 9.12 可以看出，在训练结束时，语义损失为 4.484。从图 9.14 可以看出，PSPNet 在 ADE20K 验证集上的 MIoU 约为 36.16%。

最后，利用图 9.15 的命令能够对图像语义分割的结果进行可视化，图 9.16 给出了原始图像和相应的可视化结果图像。

```

Terminal - user@ubuntu: ~/media/data/zst/PSPNet_TensorFlow
File Edit View Terminal Tabs Help
user1@ubuntu:~/media/data/zst/PSPNet_TensorFlow$ CUDA_VISIBLE_DEVICES=1 python inference.py --img-path=./input/ADE_val_00000001.jpg --dataset ade20k
successful load img: ./input/ADE_val_00000001.jpg
WARNING:tensorflow:From inference.py:90: calling argmax (from tensorflow.python.ops.math_ops) with dimension is deprecated and will be removed in a future version.
Instructions for updating:
Use the 'axis' argument instead
2018-03-29 10:50:46.677071: I tensorflow/core/platform/cpu_feature_guard.cc:137] Your CPU supports instructions that this TensorFlow binary was not compiled to use: SSE4.1 SSE4.2 AVX AVX2 FMA
2018-03-29 10:50:46.850803: I tensorflow/core/common runtime/gpu/gpu device.cc:1030] Found device 0 with properties:
name: Tesla K40c major: 3 minor: 5 memoryClockRate(GHz): 0.745
pciBusID: 0000:a1:00:0
totalMemory: 11.17GiB freeMemory: 11.03GiB
2018-03-29 10:50:46.850850: I tensorflow/core/common runtime/gpu/gpu device.cc:1120] Creating TensorFlow device (/device:GPU:0) -> (device: 0, name: Tesla K40c, pci bus id: 0000:a1:00:0, compute capability: 3.5)
Restored model parameters from ./model/model.ckpt-0
user1@ubuntu:~/media/data/zst/PSPNet_TensorFlow$

```

图 9.15 PSPNet 图像语义分割案例程序的可视化命令



图 9.16 PSPNet 图像语义分割案例程序对一个原始图像的语义分割结果

9.3 掩膜区域卷积网络 Mask R-CNN

9.3.1 Mask R-CNN 的模型结构

卷积神经网络的发展，特别是 Fast/Faster R-CNN 和 FCN^[150]，迅速提高了目标检测和语义分割的效果。目标检测是要预测图像中每个对象的位置和类别，结果一般用带有类别标记的边框来表示。语义分割则是要把每个像素都进行分类，但并不区分对象实例。一个更有



挑战性的问题是进一步把每个对象的不同实例都区分开来，称为实例分割。实例分割的难点在于，既要正确检测出图像中的所有对象，又要将不同对象精准区分开。与人们期望的复杂方法有所不同，Mask R-CNN（掩膜区域卷积网络）是一种简单、灵活、通用且快速的实例分割框架^[152]，不仅能够高效地检测目标，而且同时可以给每个实例生成一个高质量的分割掩膜。图 9.17 给出了三幅图像的实例分割结果，其中既采用边框进行定位，又区分了对象的不同实例。比如，图 9.17a 区分了不同的人，图 9.17b 区分了不同的酒杯和刀，图 9.17c 区分了不同的橘子和香蕉。

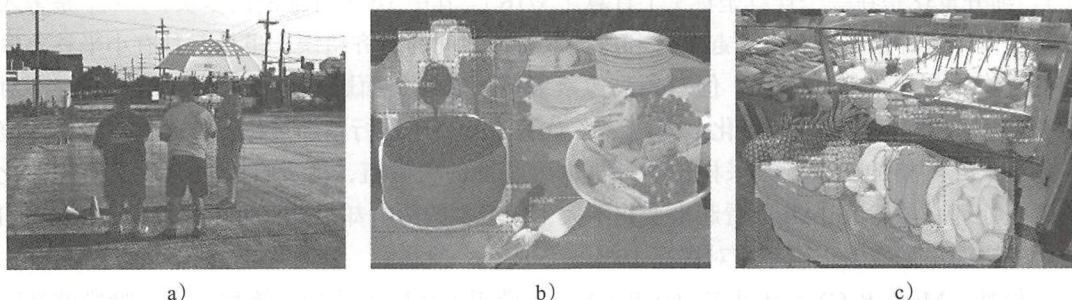


图 9.17 三幅图像的实例分割

Mask R-CNN 是在扩展 Faster R-CNN 的基础上建立起来的，关键在于创建掩膜分支。掩膜分支是一个用在 RoI 上的小型全卷积网络，与分类和边框回归分支是平行的，能够在每一个 RoI 上按像素方式预测生成高质量的分割掩膜，但增加的计算量并不大。Mask R-CNN 的优点是很容易在 Faster R-CNN 的框架上实现和训练，运行速度快，开展实验也方便。图 9.18 展示了实例分割的 Mask R-CNN 框架。

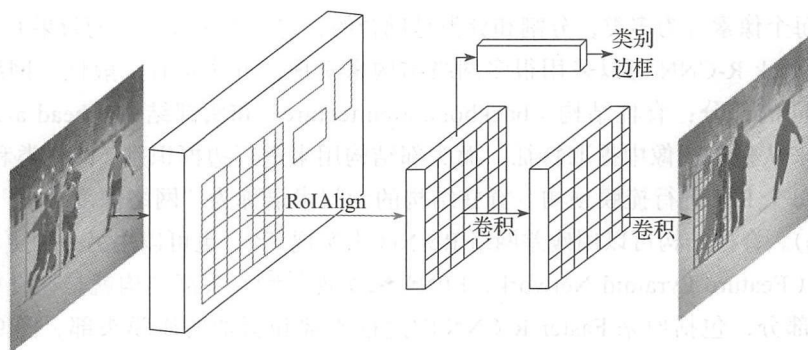


图 9.18 实例分割的 Mask R-CNN 框架

Mask R-CNN 利用掩膜对输入对象的空间布局进行编码。掩膜布局的抽取不同于类别标签和边框属性的抽取，前者可以通过卷积提供的像素级对应关系来自然完成，而后者需要通过全连接层坍塌形成的短向量来表示。具体来说，Mask R-CNN 采用一个全卷积网络预测



每个 RoI 上的 $m \times m$ 掩膜，因此可以在掩膜分支的每层都维护清晰的 $m \times m$ 对象布局，无须坍塌为没有空间维度的向量表示，但在效果上比采用全连接层的方法参数更少、准确率更高。事实上，为了更忠实地保留空间位置关系以达到更好的像素级掩膜预测，Mask R-CNN 还使用了一种对感兴趣区池化 (RoIPool) 的扩展层，称为 RoIAlign 层。与 RoIPool 相比，RoIAlign 层采用精细量化代替粗糙量化，能够在所提取特征和输入之间建立更好的对齐关系。RoIPool 是一种从 RoI 中提取小特征图 (例如 7×7 大小) 的标准操作，首先把一个浮点数 RoI 量化成离散粒度的特征图，然后再进行空间网络细量化和聚合处理 (通常是最大池化)。细量化是在一个连续坐标 x 上计算 $\lceil x/16 \rceil$ ，其中 16 是特征图的步长， $\lceil \cdot \rceil$ 是舍入取整操作。这种粗糙量化会引起 RoI 和所提取特征之间的对齐错误，虽然对具有小平移鲁棒性的分类来说可能影响不大，但对预测像素级掩膜来说负面影响是很大的。RoIAlign 不对 RoI 的边框和网格进行任何量化，采用 $x/16$ 代替 $\lceil x/16 \rceil$ 进行精细量化，并通过双线性插值在每个 RoI 网格的四个规则采样位置计算输入特征的精确值，再进行最大池化或平均池化的聚合处理。虽然 RoIAlign 看起来变化不大，但产生的影响却很大，可以修复大量 RoIPool 的对齐错误，把掩膜正确率相对提高 10% ~ 50%。

另外，Mask R-CNN 还从 Faster R-CNN 中借用了相同的两阶段流程。第一阶段就是区域推荐网络 (RPN)，用来产生候选对象边框。第二阶段平行于类别和边框预测，给每个 RoI 都输出一个二值掩膜，训练过程采用多任务损失 $L = L_{\text{cls}} + L_{\text{box}} + L_{\text{mask}}$ 。其中， L_{cls} 和 L_{box} 分别是分类损失和边框损失， L_{mask} 是掩膜损失。掩膜分支在每个 RoI 上产生一个 Km^2 维输出，为 K 个类别各编码一个 $m \times m$ 的二值掩膜。 L_{mask} 是一个用像素级 sigmoid 定义的平均二值交叉熵损失。如果一个 RoI 关联的真实类别为 k ，那么 L_{mask} 只在第 k 个掩膜上有定义，不受其他掩膜输出的影响。这种定义方式对提高实例分割的效果非常重要，它解耦了掩膜和类别的预测，无须考虑类别竞争，使 Mask R-CNN 能够独立生成每个类别的掩膜。相反，全卷积网络通常把每个像素分为多类，分割和分类是耦合的，它在实例分割上的效果并不好。

最后，Mask R-CNN 可以采用很多网络结构来实现，方法具有一般性。网络结构主要包括两个不同的部分：脊柱结构 (backbone architecture) 和头部结构 (head architecture)。脊柱结构用来从整幅图像中提取特征，而头部结构用来进行边框识别 (即分类和回归)，并单独用来对每个 RoI 进行掩膜预测。脊柱结构的专门术语称为“网络深度特征” (network-depth-feature)。脊柱结构可以用残差网络 ResNet 来实现^[68]，也可以用另一种更有效的特征金字塔网络 (Feature Pyramid Network, FPN) 来实现^[153]。头部结构就是在脊柱结构之上增加的分支部分，包括原来 Faster R-CNN 的边框头部和新增的掩膜头部。图 9.19 针对用 ResNet 和 FPN 实现的 Faster R-CNN，分别说明了头部结构的情况。

在训练时，Mask R-CNN 的超参数是根据 Fast/Faster R-CNN 的选择来设置的，同样的设置对实例分割的鲁棒性也非常好。如果一个 RoI 与真实边框的交并比大于等于 0.5，就被认为是正例，否则认为是反例。掩膜损失 L_{mask} 只在正例上有定义，其目标是最大化与真实掩膜的交集。所有图像在训练前要进行尺度大小调整，使短边为 800 像素。每个迷你块在



每个 GPU 上分配 2 幅图像, 在每幅图像上按正反比例 1 : 3 采样 N 个 RoI。 N 的取值与脊柱结构的选择有关, 比如对 ResNet 结构可设置 $N = 64$, 对 FPN 结构可设置 $N = 512$ 。在 8 个 GPU 上的有效迷你块大小为 16, 达到预期效果需要训练 160 000 次迭代, 学习率为 0.02, 在 120 000 次迭代时衰减到原来的 1/10。另外, 权值衰减系数取为 0.0001, 动量项系数取为 0.9。

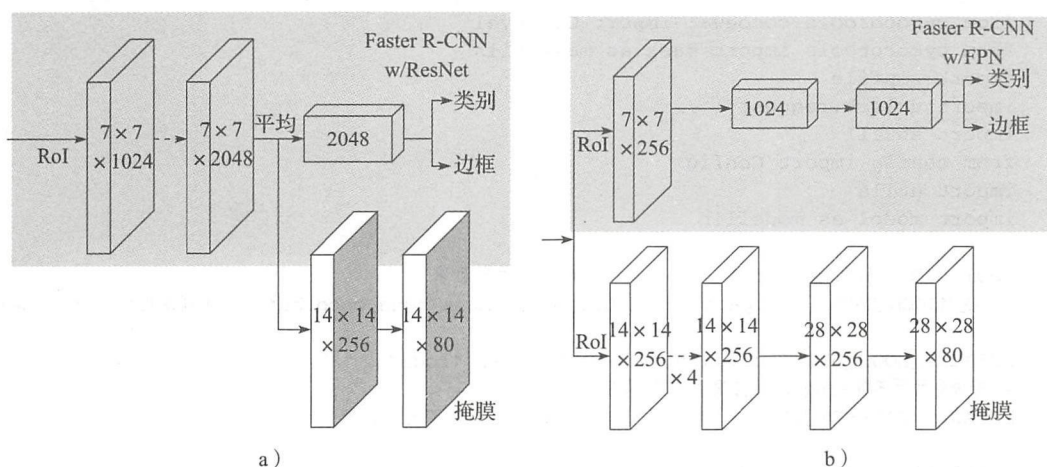


图 9.19 两种不同的头部结构, 其中 Faster R-CNN 的脊柱结构分别是 a) ResNet 和 b) FPN。数字代表空间分辨率和通道数, 箭头代表卷积、反卷积或全连接层。除了输出卷积是 1×1 之外, 所有其他卷积都是 3×3 , 反卷积的核大小是 2×2 , 步长是 2。隐含层的激活函数采用 ReLU。“res5”表示 ResNet 的第 5 阶段, “ $\times 4$ ”表示 4 个相继卷积的堆叠

在测试时, 推荐候选区的数量也和脊柱结构有关, 对 ResNet 结构推荐 300 个, 对 FPN 结构推荐 1000 个。每个推荐区域都要运行边框预测分支, 然后进行非极大值抑制。掩膜分支只应用于得分最高的 100 个检测边框, 目的是利用更少更精的 RoI 加快推断速度和提高正确率。掩膜分支能够给每个 RoI 预测 K 个掩膜, 但只用根据分类分支预测决定的第 k 个掩膜。最后, $m \times m$ 浮点数大小的掩膜输出被归一化到 RoI 的大小, 并以 0.5 为阈值进行二值化。由于 Mask R-CNN 只在前 100 个检测边框上计算掩膜, 所以增加的边际运行时间并不多, 与相应的 Faster R-CNN 相比, 大约增加 20%。

9.3.2 Mask R-CNN 的 Keras 和 TensorFlow 代码实现及说明

关于 Mask R-CNN 的 Keras 和 TensorFlow 代码, 下载地址为 https://github.com/matterport/Mask_RCNN。这个网址中的文件夹内包含多个文件和文件夹, 例如求解器配置文件 coco.py、模型定义文件 model.py 和可视化文件 visualize.py 等, 以及存放数据集文件夹 path、存放权值文件夹 logs 和可视化图像文件夹 images 等。考虑到有些代码与之前的模型有重复, 下面仅对求解器配置文件 coco.py 和模型定义文件 model.py 分别进行详细介绍和说明。



1. 求解器配置文件 coco.py 代码及说明

```
import os                # 导入 os 模块
import time              # 导入 time 模块
import numpy as np       # 导入 numpy 模块，并以 np 作为别名

# Download and install the Python COCO tools from https://github.com/waleedka/coco
from pycocotools.coco import COCO      # 从 pycocotools.coco 模块中导入 COCO
from pycocotools.cocoeval import COCOeval
from pycocotools import mask as maskUtils
import zipfile
import urllib.request
import shutil
from config import Config
import utils
import model as modellib

ROOT_DIR = os.getcwd()          # 工程的根目录
COCO_MODEL_PATH = os.path.join(ROOT_DIR, "mask_rcnn_coco.h5")  # 训练好的权值文件路径

DEFAULT_LOGS_DIR = os.path.join(ROOT_DIR, "logs")
# 若命令行无参数 -logs, 则在根目录下保存
DEFAULT_DATASET_YEAR = "2014"   # 默认 2014 年的数据集

#####
# 在 MS COCO 上训练配置
#####
class CocoConfig(Config):
    NAME = "coco"                # 赋予一个可识别的配置名
    IMAGES_PER_GPU = 2          # 设置每个 GPU 运行的图像数
    # GPU_COUNT = 8              # 设置所用 GPU 的个数，默认为 1 个，如果删除注释符 #，则使用 8 个
    NUM_CLASSES = 1 + 80        # COCO 包含 80 个物体类别和 1 个背景类别

#####
# 数据集
#####
class CocoDataset(utils.Dataset):
    def load_coco(self, dataset_dir, subset, year=DEFAULT_DATASET_YEAR, class_ids=None,
class_map=None, return_coco=False, auto_download=False):          # 导入 COCO 数据集
        if auto_download is True:  # 如果 auto_download 为真，则自动下载数据集
            self.auto_download(dataset_dir, subset, year)
            coco = COCO("{} / annotations / instances_{} .json".format(dataset_dir,
subset, year))
        if subset == "minival" or subset == "valminusminival":
            subset = "val"
        image_dir = "{} / {} {}".format(dataset_dir, subset, year)
        if not class_ids:  # 如果没有提供类别编号，则导入所有的类别
            class_ids = sorted(coco.getCatIds())
        if class_ids:      # 如果提供类别编号，则导入包含类别编号的图像
            image_ids = []
            for id in class_ids:
                image_ids.extend(list(coco.getImgIds(catIds=[id])))
```



```

        image_ids = list(set(image_ids)) # 删除重复的图像编号
    else:
        image_ids = list(coco.imgs.keys())
        for i in class_ids: # 增加类别
            self.add_class("coco", i, coco.loadCats(i)[0]["name"])

    for i in image_ids: # 增加所有图像
        self.add_image("coco", image_id=i,
            path=os.path.join(image_dir, coco.imgs[i]['file_name']),
            width=coco.imgs[i]["width"],
            height=coco.imgs[i]["height"],
            annotations=coco.loadAnns(coco.getAnnIds(
                imgIds=[i], catIds=class_ids, iscrowd=None)))
    if return_coco:
        return coco

def auto_download(self, dataDir, dataType, dataYear): # 下载 COCO 数据集
    if dataType == "minival" or dataType == "valminusminival":
        # 设置路径和文件名字
        imgDir = "{}/{}/{}/".format(dataDir, "val", dataYear)
        imgZipFile = "{}/{}/{}/.zip".format(dataDir, "val", dataYear)
        imgURL = "http://images.cocodataset.org/zips/{}/{}/.zip".format("val", dataYear)
    else:
        imgDir = "{}/{}/{}/".format(dataDir, dataType, dataYear)
        imgZipFile = "{}/{}/{}/.zip".format(dataDir, dataType, dataYear)
        imgURL = "http://images.cocodataset.org/zips/{}/{}/.zip".format(dataType,
            dataYear)

    if not os.path.exists(dataDir): # 如果没有主文件夹, 那么创建它
        os.makedirs(dataDir)

    if not os.path.exists(imgDir): # 如果本地不能获取图像文件, 那么进行下载
        os.makedirs(imgDir)
        print("Downloading images to " + imgZipFile + " ...")
        with urllib.request.urlopen(imgURL) as resp, open(imgZipFile, 'wb')
        as out: shutil.copyfileobj(resp, out)
        print("... done downloading.")
        print("Unzipping " + imgZipFile)
        with zipfile.ZipFile(imgZipFile, "r") as zip_ref:
            zip_ref.extractall(dataDir)
        print("... done unzipping")
    print("Will use images in " + imgDir)

    annDir = "{}/{}/annotations".format(dataDir) # 创建标注数据路径
    if dataType == "minival":
        annZipFile = "{}/{}/instances_minival2014.json.zip".format(dataDir)
        annFile = "{}/{}/instances_minival2014.json".format(annDir)
        annURL = "https://dl.dropboxusercontent.com/s/o43o90bna78omob/instances_\
            minival2014.json.zip?dl=0"
        unZipDir = annDir
    elif dataType == "valminusminival":
        annZipFile = "{}/{}/instances_valminusminival2014.json.zip".format(dataDir)

```



```

annFile = "{}/instances_valminusminival2014.json".format(annDir)
annURL = "https://dl.dropboxusercontent.com/s/s3tw5zcg7395368/instances_valminusminival2014.json.zip?dl=0"
unZipDir = annDir
else:
    annZipFile = "{}/annotations_trainval{}.zip".format(dataDir, dataYear)
    annFile = "{}/instances_{}.json".format(annDir, dataYear)
    annURL = "http://images.cocodataset.org/annotations/annotations_trainval{}.zip".format(dataYear)
    unZipDir = dataDir

if not os.path.exists(annDir):      # 没有标签数据，那么创建它
    os.makedirs(annDir)
if not os.path.exists(annFile):
    if not os.path.exists(annZipFile):
        print("Downloading zipped annotations to " + annZipFile + " ...")
        with urllib.request.urlopen(annURL) as resp, open(annZipFile, 'wb') as out:
            shutil.copyfileobj(resp, out)
        print("... done downloading.")
        print("Unzipping " + annZipFile)
        with zipfile.ZipFile(annZipFile, "r") as zip_ref:
            zip_ref.extractall(unZipDir)
        print("... done unzipping")
    print("Will use annotations in " + annFile)

def load_mask(self, image_id):      # 对给定的图像导入掩膜
    image_info = self.image_info[image_id]
    if image_info["source"] != "coco":
        # 判断是否为 COCO 图像，如果不是，则将其委托给父类
        return super(CocoDataset, self).load_mask(image_id)

    instance_masks = []
    class_ids = []
    annotations = self.image_info[image_id]["annotations"]
    # Build mask of shape [height, width, instance_count] and list
    # of class IDs that correspond to each channel of the mask.
    # 创建形状为 [height, width, instance_count] 的掩码，并为掩码的每个通道的
    # 类别 ID 创建列表
    for annotation in annotations:
        class_id = self.map_source_class_id("coco.{}".format(annotation['category_id']))
        if class_id:
            m = self.annToMask(annotation, image_info["height"], image_info["width"])
            if m.max() < 1:      # 忽略像素面积小于 1 的物体
                continue
            if annotation['iscrowd']: # 如果是密集区域，则使用负例 ID
                class_id *= -1
            if m.shape[0] != image_info["height"] or m.shape[1] != image_info["width"]:
                m = np.ones([image_info["height"], image_info["width"]], dtype=bool)

```




```

        instance_masks.append(m)
        class_ids.append(class_id)

    if class_ids:
        # 若将实例掩膜封装到一个数组中
        mask = np.stack(instance_masks, axis=2)
        class_ids = np.array(class_ids, dtype=np.int32)
        return mask, class_ids
    else:
        return super(CocoDataset, self).load_mask(image_id)
        # 否则, 调用超类返回一个空掩膜

def image_reference(self, image_id):
    # 返回图像在 COCO 网站上的链接
    info = self.image_info[image_id]
    if info["source"] == "coco":
        return "http://cocodataset.org/#explore?id={}".format(info["id"])
    else:
        super(CocoDataset, self).image_reference(image_id)

def annToRLE(self, ann, height, width):
    # 将无压缩编码的标注转换为 RLE 编码
    segm = ann['segmentation']
    if isinstance(segm, list):
        rles = maskUtils.frPyObjects(segm, height, width)
        rle = maskUtils.merge(rles)
    elif isinstance(segm['counts'], list):
        rle = maskUtils.frPyObjects(segm, height, width)
    else:
        rle = ann['segmentation']
    return rle

def annToMask(self, ann, height, width):
    # 将无压缩的或者 RLE 编码的标注转换为二值掩膜
    rle = self.annToRLE(ann, height, width)
    m = maskUtils.decode(rle)
    return m

#####
# COCO 验证
#####
def build_coco_results(dataset, image_ids, rois, class_ids, scores, masks):
    # 将结果整理成 COCO 网站上的格式
    if rois is None:
        # 如果没有结果, 那么返回一个空列表
        return []

    results = []
    for image_id in image_ids:
        for i in range(rois.shape[0]):
            class_id = class_ids[i]
            score = scores[i]
            bbox = np.around(rois[i], 1)
            mask = masks[:, :, i]
            result = {
                "image_id": image_id,

```



```

        "category_id": dataset.get_source_class_id(class_id, "coco"),
        "bbox": [bbox[1], bbox[0], bbox[3] - bbox[1], bbox[2] - bbox[0]],
        "score": score,
        "segmentation": maskUtils.encode(np.asfortranarray(mask))
    }
    results.append(result)
return results

def evaluate_coco(model, dataset, coco, eval_type="segm", limit=0, image_ids=None):
    # 使用验证集进行验证
    image_ids = image_ids or dataset.image_ids # 选择 COCO 图像
    if limit: # 如果限定了子集, 那么只选择子集中的图像
        image_ids = image_ids[:limit]
    coco_image_ids = [dataset.image_info[id]["id"] for id in image_ids]
    t_prediction = 0
    t_start = time.time()

    results = []
    for i, image_id in enumerate(image_ids):
        image = dataset.load_image(image_id) # 导入图像
        t = time.time()
        r = model.detect([image], verbose=0)[0] # 进行目标检测
        t_prediction += (time.time() - t)
        image_results = build_coco_results(dataset, coco_image_ids[i:i + 1],
            r["rois"], r["class_ids"], r["scores"], r["masks"])
        results.extend(image_results)
    coco_results = coco.loadRes(results)

    cocoEval = COCOeval(coco, coco_results, eval_type) # 对结果进行统计
    cocoEval.params.imgIds = coco_image_ids
    cocoEval.evaluate()
    cocoEval.accumulate()
    cocoEval.summarize()

    print("Prediction time: {}. Average {}/image".format(t_prediction, t_prediction / \
        len(image_ids)))
    print("Total time: ", time.time() - t_start)

#####
# 训练
#####
if __name__ == '__main__':
    import argparse

    # 解析命令行参数
    parser = argparse.ArgumentParser(description='Train Mask R-CNN on MS COCO.')
    parser.add_argument("command", metavar="<command>", help="'train' or 'evaluate'
        on MS COCO")
    parser.add_argument('--dataset', required=True, metavar="/path/to/coco",
        help='Directory of the MS-COCO dataset')
    parser.add_argument('--year', required=False, default=DEFAULT_DATASET_YEAR,
        metavar="<year>", help='Year of the MS-COCO dataset (2014 or 2017)')

```

```

(default=2014)')
parser.add_argument('--model', required=True, metavar="/path/to/weights.h5",
                    help="Path to weights .h5 file or 'coco'")
parser.add_argument('--logs', required=False, default=DEFAULT_LOGS_DIR,
                    metavar="/path/to/logs/", help='Logs and checkpoints directory'
                    (default=logs/))
parser.add_argument('--limit', required=False, default=500,
                    metavar="<image count>", help='Images to use for evaluation (default=500)')
parser.add_argument('--download', required=False, default=False, metavar=
"<True|False>", help='Automatically download and unzip MS-COCO files (default=False)',
                    type=bool)
args = parser.parse_args()
print("Command: ", args.command)
print("Model: ", args.model)
print("Dataset: ", args.dataset)
print("Year: ", args.year)
print("Logs: ", args.logs)
print("Auto Download: ", args.download)

if args.command == "train":
    # 参数配置
    config = CocoConfig()
else:
    class InferenceConfig(CocoConfig):
        GPU_COUNT = 1
        IMAGES_PER_GPU = 1
        DETECTION_MIN_CONFIDENCE = 0
    config = InferenceConfig()
config.display()

if args.command == "train":
    # 创建模型
    model = modellib.MaskRCNN(mode="training", config=config, model_dir=args.\
logs)
else:
    model = modellib.MaskRCNN(mode="inference", config=config, model_dir=args.\
logs)

if args.model.lower() == "coco":
    # 选择权值文件
    model_path = COCO_MODEL_PATH
elif args.model.lower() == "last":
    model_path = model.find_last()[1]
elif args.model.lower() == "imagenet":
    model_path = model.get_imagenet_weights()
else:
    model_path = args.model

print("Loading weights ", model_path)
model.load_weights(model_path, by_name=True) # 导入权值文件

if args.command == "train":
    dataset_train = CocoDataset()
    dataset_train.load_coco(args.dataset, "train", year=args.year, auto_down=\
load=args.download)

```



```

dataset_train.load_coco(args.dataset, "valminusminival", year=args.year,
auto_download=args.download)
dataset_train.prepare()
dataset_val = CocoDataset()
dataset_val.load_coco(args.dataset, "minival", year=args.year, auto_down-
load=args.download)
dataset_val.prepare()

print("Training network heads") # 第1个训练阶段
model.train(dataset_train, dataset_val, learning_rate=config.LEARNING_
RATE,
epochs=40, layers='heads')

print("Fine tune Resnet stage 4 and up") # 第2个训练阶段
model.train(dataset_train, dataset_val, learning_rate=config.LEARNING_
RATE, epochs=120, layers='4+')

print("Fine tune all layers") # 第3个训练阶段
model.train(dataset_train, dataset_val, learning_rate=config.LEARNING_
RATE / 10, epochs=160, layers='all')

elif args.command == "evaluate":
dataset_val = CocoDataset()
coco = dataset_val.load_coco(args.dataset, "minival", year=args.year,
return_coco=True, auto_download=args.download)
dataset_val.prepare()
print("Running COCO evaluation on {} images.".format(args.limit))
evaluate_coco(model, dataset_val, coco, "bbox", limit=int(args.limit))
else:
print('{}' is not recognized. " "Use 'train' or 'evaluate'".format(args.\
command))

```

2. 模型定义文件 model.py 代码及详细说明

模型定义文件 model.py 包含多个模块，主要有残差图（resnet graph）模块、推荐层（proposal layer）、RoIAlign 层（RoIAlign layer）、检测目标层（detection target layer）、检测层（detection layer）、特征金字塔网络头部（feature pyramid network head）、损失函数（loss function）、数据生成器（data generator）和 MaskRCNN 类（MaskRCNN class）等。考虑到有些代码与 Faster R-CNN 存在重复，下面仅给出 RoIAlign 层、特征金字塔网络头部、损失函数和 MaskRCNN 类这四部分的代码及详细说明。

（1）RoIAlign 层代码及详细说明

```

#####
# ROIAlign Layer ROIAlign 层
#####
def log2_graph(x):
    return tf.log(x) / tf.log(2.0)

class PyramidROIAlign(KE.Layer): # 在特征金字塔的多个层级实现 RoI 池化

```

```

def __init__(self, pool_shape, image_shape, **kwargs):
    super(PyramidROIAlign, self).__init__(**kwargs)
    self.pool_shape = tuple(pool_shape)
    self.image_shape = tuple(image_shape)

def call(self, inputs): # 在归一化坐标系中裁剪边框
    [batch, num_boxes, (y1, x1, y2, x2)]
    boxes = inputs[0]
    feature_maps = inputs[1:]
    y1, x1, y2, x2 = tf.split(boxes, 4, axis=2)
    # 依据 RoI, 为每个 RoI 分配一个金字塔层级
    h = y2 - y1
    w = x2 - x1
    image_area = tf.cast(self.image_shape[0] * self.image_shape[1], tf.float32)
    roi_level = log2_graph(tf.sqrt(h * w) / (224.0 / tf.sqrt(image_area)))
    roi_level = tf.minimum(5, tf.maximum(2, 4 + tf.cast(tf.round(roi_level),
    tf.int32)))
    roi_level = tf.squeeze(roi_level, 2)

    pooled = []
    box_to_level = []
    for i, level in enumerate(range(2, 6)): # 遍历层级
        ix = tf.where(tf.equal(roi_level, level))
        level_boxes = tf.gather_nd(boxes, ix)
        box_indices = tf.cast(ix[:, 0], tf.int32)
        box_to_level.append(ix) # 跟踪哪个边框被映射到哪个层级
        level_boxes = tf.stop_gradient(level_boxes)
        box_indices = tf.stop_gradient(box_indices)
        pooled.append(tf.image.crop_and_resize(feature_maps[i], level_boxes,
        box_indices, self.pool_shape, method="bilinear"))
    pooled = tf.concat(pooled, axis=0) # 将池化特征组合成一个张量

    box_to_level = tf.concat(box_to_level, axis=0)
    box_range = tf.expand_dims(tf.range(tf.shape(box_to_level)[0]), 1)
    box_to_level = tf.concat([tf.cast(box_to_level, tf.int32), box_range],
    axis=1)

    sorting_tensor = box_to_level[:, 0] * 100000 + box_to_level[:, 1]
    # 重新组合池化特征以匹配原始边框
    ix = tf.nn.top_k(sorting_tensor, k=tf.shape(box_to_level)[0]).indices[::-1]
    ix = tf.gather(box_to_level[:, 2], ix)
    pooled = tf.gather(pooled, ix)
    pooled = tf.expand_dims(pooled, 0) # 重新增加块维度
    return pooled

```

(2) 特征金字塔网络头部代码及详细说明

```

#####
# 特征金字塔网络头部
#####
# 建立特征金字塔网络分类器和回归器的头部
def fpn_classifier_graph(rois, feature_maps, image_shape, pool_size, num_classes):
    x = PyramidROIAlign([pool_size, pool_size], image_shape, name="roi_align_classifier")

```

```

([rois] + feature_maps)
x = KL.TimeDistributed(KL.Conv2D(1024, (pool_size, pool_size), padding="valid"),
                        name="mrcnn_class_conv1")(x)
x = KL.TimeDistributed(BatchNorm(axis=3), name='mrcnn_class_bn1')(x)
x = KL.Activation('relu')(x)
x = KL.TimeDistributed(KL.Conv2D(1024, (1, 1)), name="mrcnn_class_conv2")(x)
x = KL.TimeDistributed(BatchNorm(axis=3), name='mrcnn_class_bn2')(x)
x = KL.Activation('relu')(x)
shared = KL.Lambda(lambda x: K.squeeze(K.squeeze(x, 3), 2), name="pool_squeeze")(x)

# 分类器头部
mrcnn_class_logits = KL.TimeDistributed(KL.Dense(num_classes), name='mrcnn\
class_logits')(shared)
mrcnn_probs = KL.TimeDistributed(KL.Activation("softmax"), name="mrcnn_class")
(mrcnn_class_logits)

# BBox 头部
x = KL.TimeDistributed(KL.Dense(num_classes * 4, activation='linear'), name=\
'mrcnn_bbox_fc')(shared)
s = K.int_shape(x)
mrcnn_bbox = KL.Reshape((s[1], num_classes, 4), name="mrcnn_bbox")(x)

return mrcnn_class_logits, mrcnn_probs, mrcnn_bbox

# 建立特征金字塔网络的掩膜头部
def build_fpn_mask_graph(rois, feature_maps, image_shape, pool_size, num_classes):
    x = PyramidROIAlign([pool_size, pool_size], image_shape, name="roi_align\
mask")([rois] + feature_maps)

    # 卷积层
    x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"), name="mrcnn\
mask_conv1")(x)
    x = KL.TimeDistributed(BatchNorm(axis=3), name='mrcnn_mask_bn1')(x)
    x = KL.Activation('relu')(x)

    x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"), name="mrcnn\
mask_conv2")(x)
    x = KL.TimeDistributed(BatchNorm(axis=3), name='mrcnn_mask_bn2')(x)
    x = KL.Activation('relu')(x)

    x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"), name="mrcnn\
mask_conv3")(x)
    x = KL.TimeDistributed(BatchNorm(axis=3), name='mrcnn_mask_bn3')(x)
    x = KL.Activation('relu')(x)

    x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"), name="mrcnn\
mask_conv4")(x)
    x = KL.TimeDistributed(BatchNorm(axis=3), name='mrcnn_mask_bn4')(x)
    x = KL.Activation('relu')(x)

    x = KL.TimeDistributed(KL.Conv2DTranspose(256, (2, 2), strides=2, activation="relu"),
name="mrcnn_mask_deconv")(x)
    x = KL.TimeDistributed(KL.Conv2D(num_classes, (1, 1), strides=1, activation=\

```



```

"sigmoid"),
                                name="mrcnn_mask")(x)
return x

```

(3) 损失函数代码及详细说明

```

#####
# 损失函数
#####
def smooth_l1_loss(y_true, y_pred): # 实现平滑-L1 损失
    diff = K.abs(y_true - y_pred)
    less_than_one = K.cast(K.less(diff, 1.0), "float32")
    loss = (less_than_one * 0.5 * diff**2) + (1 - less_than_one) * (diff - 0.5)
    return loss

def rpn_class_loss_graph(rpn_match, rpn_class_logits) : # RPN 锚点分类器损失
    rpn_match = tf.squeeze(rpn_match, -1)
    anchor_class = K.cast(K.equal(rpn_match, 1), tf.int32)
    # 获得锚点类别, 将 -1/+1 匹配转换到 0/1 值
# 正锚点和负锚点贡献损失, 而中立的锚点不贡献损失
    indices = tf.where(K.not_equal(rpn_match, 0))
    rpn_class_logits = tf.gather_nd(rpn_class_logits, indices)
    anchor_class = tf.gather_nd(anchor_class, indices)
    # 交叉熵损失
    loss = K.sparse_categorical_crossentropy(target=anchor_class, output=rpn_\
class_logits, from_logits=True)
    loss = K.switch(tf.size(loss) > 0, K.mean(loss), tf.constant(0.0))
    return loss

def rpn_bbox_loss_graph(config, target_bbox, rpn_match, rpn_bbox):
# 返回 RPN 边框损失
    rpn_match = K.squeeze(rpn_match, -1)
    indices = tf.where(K.equal(rpn_match, 1))

    rpn_bbox = tf.gather_nd(rpn_bbox, indices) # 选择贡献损失的边框
    batch_counts = K.sum(K.cast(K.equal(rpn_match, 1), tf.int32), axis=1)
    target_bbox = batch_pack_graph(target_bbox, batch_counts, config.IMAGES_PER_GPU)

    diff = K.abs(target_bbox - rpn_bbox)
    less_than_one = K.cast(K.less(diff, 1.0), "float32")
    loss = (less_than_one * 0.5 * diff**2) + (1 - less_than_one) * (diff - 0.5)
    loss = K.switch(tf.size(loss) > 0, K.mean(loss), tf.constant(0.0))
    return loss

def mrcnn_class_loss_graph(target_class_ids, pred_class_logits, active_class_ids):
                                # 掩膜 R-CNN 分类器头部损失
    target_class_ids = tf.cast(target_class_ids, 'int64')
    pred_class_ids = tf.argmax(pred_class_logits, axis=2)
    pred_active = tf.gather(active_class_ids[0], pred_class_ids)

    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=target_class_\
ids, logits=pred_class_logits)
    loss = loss * pred_active

```

```

    loss = tf.reduce_sum(loss) / tf.reduce_sum(pred_active) # 计算损失均值
    return loss

def mrcnn_bbox_loss_graph(target_bbox, target_class_ids, pred_bbox):
# Mask R-CNN 边框损失
    target_class_ids = K.reshape(target_class_ids, (-1,))
    target_bbox = K.reshape(target_bbox, (-1, 4))
    pred_bbox = K.reshape(pred_bbox, (-1, K.int_shape(pred_bbox)[2], 4))

    positive_roi_ix = tf.where(target_class_ids > 0)[: , 0] # 只有正的 RoI 贡献损失
    positive_roi_class_ids = tf.cast(tf.gather(target_class_ids, positive_roi_ix), tf.int64)
    indices = tf.stack([positive_roi_ix, positive_roi_class_ids], axis=1)

    target_bbox = tf.gather(target_bbox, positive_roi_ix) # 收集贡献损失的增量
    pred_bbox = tf.gather_nd(pred_bbox, indices)

    loss = K.switch(tf.size(target_bbox) > 0, smooth_l1_loss(y_true=target_bbox,
y_pred=pred_bbox), tf.constant(0.0))
    loss = K.mean(loss)
    loss = K.reshape(loss, [1, 1])
    return loss

def mrcnn_mask_loss_graph(target_masks, target_class_ids, pred_masks):
# 用于掩膜头部的二值交叉熵损失
    target_class_ids = K.reshape(target_class_ids, (-1,))
    mask_shape = tf.shape(target_masks)
    target_masks = K.reshape(target_masks, (-1, mask_shape[2], mask_shape[3]))
    pred_shape = tf.shape(pred_masks)
    pred_masks = K.reshape(pred_masks, (-1, pred_shape[2], pred_shape[3], pred_shape[4]))
    pred_masks = tf.transpose(pred_masks, [0, 3, 1, 2])

    positive_ix = tf.where(target_class_ids > 0)[: , 0]
    positive_class_ids = tf.cast(tf.gather(target_class_ids, positive_ix), tf.int64)
    indices = tf.stack([positive_ix, positive_class_ids], axis=1)

    y_true = tf.gather(target_masks, positive_ix)
    y_pred = tf.gather_nd(pred_masks, indices)

    loss = K.switch(tf.size(y_true) > 0, K.binary_crossentropy(target=y_true,
output=y_pred), tf.constant(0.0)) # 计算二值交叉熵。如果没有正的 RoI, 那么返回 0
    loss = K.mean(loss)
    loss = K.reshape(loss, [1, 1])
    return loss

```

(4) Mask R-CNN 类代码及详细说明

```

#####
# MaskRCNN 类
#####
class MaskRCNN(): # 封装 Mask R-CNN 模型的功能
    def __init__(self, mode, config, model_dir):

```

```

assert mode in ['training', 'inference']
self.mode = mode
self.config = config
self.model_dir = model_dir
self.set_log_dir()
self.keras_model = self.build(mode=mode, config=config)

def build(self, mode, config):    # 创建Mask R-CNN的结构
    assert mode in ['training', 'inference']
    h, w = config.IMAGE_SHAPE[:2] # 图像大小必须为2的倍数
    if h / 2**6 != int(h / 2**6) or w / 2**6 != int(w / 2**6):
        raise Exception("Image size must be dividable by 2 at least 6 times "
                          "to avoid fractions when downscaling and upscaling."
                          "For example, use 256, 320, 384, 448, 512, ... etc. ")

    input_image = KL.Input(shape=config.IMAGE_SHAPE.tolist(), name="input_\
image") # 输入
    input_image_meta = KL.Input(shape=[None], name="input_image_meta")
    if mode == "training":
        input_rpn_match = KL.Input(shape=[None, 1], name="input_rpn_match",
                                     dtype=tf.int32)
        input_rpn_bbox = KL.Input(shape=[None, 4], name="input_rpn_bbox",
                                    dtype=tf.float32)

        # 检测真实信息 GT (class IDs, bounding boxes, and masks)
        # 1. GT Class IDs (zero padded)
        input_gt_class_ids = KL.Input(shape=[None], name="input_gt_class_\
ids", dtype=tf.int32)
        # 2. GT Boxes in pixels (zero padded)
        input_gt_boxes = KL.Input(shape=[None, 4], name="input_gt_boxes",
                                   dtype=tf.float32)
        h, w = K.shape(input_image)[1], K.shape(input_image)[2]
        image_scale = K.cast(K.stack([h, w, h, w], axis=0), tf.float32)
        gt_boxes = KL.Lambda(lambda x: x / image_scale)(input_gt_boxes)
        # 3. GT Masks (zero padded)
        if config.USE_MINI_MASK:
            input_gt_masks = KL.Input(shape=[config.MINI_MASK_SHAPE[0],
                                              config.MINI_MASK_SHAPE[1], None],
                                       name="input_gt_masks", dtype=bool)
        else:
            input_gt_masks = KL.Input(shape=[config.IMAGE_SHAPE[0], config.\
IMAGE_SHAPE[1], None], name="input_gt_masks", dtype=bool)

    # 创建共享卷积层
    _, C2, C3, C4, C5 = resnet_graph(input_image, "resnet101", stage5=True)
    P5 = KL.Conv2D(256, (1, 1), name='fpn_c5p5')(C5)
    P4 = KL.Add(name="fpn_p4add")((KL.UpSampling2D(size=(2, 2), name="fpn_\
p5upsampled")(P5), KL.Conv2D(256, (1, 1), name='fpn_c4p4')(C4))
    P3 = KL.Add(name="fpn_p3add")((KL.UpSampling2D(size=(2, 2), name="fpn_\
p4upsampled")(P4),
    KL.Conv2D(256, (1, 1), name='fpn_c3p3')(C3))
    P2 = KL.Add(name="fpn_p2add")((KL.UpSampling2D(size=(2, 2), name="fpn_\
p3upsampled")(P3),

```



```

        KL.Conv2D(256, (1, 1), name='fpn_c2p2')(C2)])
P2 = KL.Conv2D(256, (3, 3), padding="SAME", name="fpn_p2")(P2)
P3 = KL.Conv2D(256, (3, 3), padding="SAME", name="fpn_p3")(P3)
P4 = KL.Conv2D(256, (3, 3), padding="SAME", name="fpn_p4")(P4)
P5 = KL.Conv2D(256, (3, 3), padding="SAME", name="fpn_p5")(P5)

# P6 用于 RPN 的第 5 个锚点尺度，它是由 P5 使用步长为 2 进行下采样得到的。
P6 = KL.MaxPooling2D(pool_size=(1, 1), strides=2, name="fpn_p6")(P5)

rpn_feature_maps = [P2, P3, P4, P5, P6]
mrcnn_feature_maps = [P2, P3, P4, P5]

self.anchors = utils.generate_pyramid_anchors(config.RPN_ANCHOR_SCALES,
                                              config.RPN_ANCHOR_RATIOS,
                                              config.BACKBONE_SHAPES,
                                              config.BACKBONE_STRIDES,
                                              config.RPN_ANCHOR_STRIDE)

# 生成锚点

# RPN 模型
rpn = build_rpn_model(config.RPN_ANCHOR_STRIDE, len(config.RPN_ANCHOR_
RATIOS), 256)
layer_outputs = []
for p in rpn_feature_maps:
    layer_outputs.append(rpn([p]))
output_names = ["rpn_class_logits", "rpn_class", "rpn_bbox"]
outputs = list(zip(*layer_outputs))
outputs = [KL.Concatenate(axis=1, name=n)(list(o))
           for o, n in zip(outputs, output_names)]

rpn_class_logits, rpn_class, rpn_bbox = outputs

proposal_count = config.POST_NMS_ROIS_TRAINING if mode == "training" \
# 生成推荐
else config.POST_NMS_ROIS_INFERENCE
rpn_rois = ProposalLayer(proposal_count=proposal_count,
                        nms_threshold=config.RPN_NMS_THRESHOLD,
                        name="ROI", anchors=self.anchors,
config=config)([rpn_class, rpn_bbox])

if mode == "training":
    _, _, _, active_class_ids = KL.Lambda(lambda x: parse_image_meta_
graph(x), mask=[None, None, None, None])(input_image_meta)

    if not config.USE_RPN_ROIS:
        input_rois = KL.Input(shape=[config.POST_NMS_ROIS_TRAINING, 4],
                              name="input_roi", dtype=np.int32)
        target_rois = KL.Lambda(lambda x: K.cast(x, tf.float32) / image_scale\
[:4])(input_rois)
    else:
        target_rois = rpn_rois

```

```

rois, target_class_ids, target_bbox, target_mask = \
    DetectionTargetLayer(config, name="proposal_targets")([
        target_rois, input_gt_class_ids, gt_boxes, input_gt_masks])
    # 生成检测目标

mrcnn_class_logits, mrcnn_class, mrcnn_bbox = \
    fpn_classifier_graph(rois, mrcnn_feature_maps, config.IMAGE_SHAPE,
                        config.POOL_SIZE, config.NUM_CLASSES)

    # 网络头部
mrcnn_mask = build_fpn_mask_graph(rois, mrcnn_feature_maps, config.\
IMAGE_SHAPE, config.MASK_POOL_SIZE, config.NUM_CLASSES)
output_rois = KL.Lambda(lambda x: x * 1, name="output_rois")(rois)

# 计算损失
rpn_class_loss = KL.Lambda(lambda x: rpn_class_loss_graph(*x), name=\
"rpn_class_loss")(
    [input_rpn_match, rpn_class_logits])
rpn_bbox_loss = KL.Lambda(lambda x: rpn_bbox_loss_graph(config, *x),
name="rpn_bbox_loss")(
    [input_rpn_bbox, input_rpn_match, rpn_bbox])
class_loss = KL.Lambda(lambda x: mrcnn_class_loss_graph(*x), name=\
"mrcnn_class_loss")(
    [target_class_ids, mrcnn_class_logits, active_class_ids])
bbox_loss = KL.Lambda(lambda x: mrcnn_bbox_loss_graph(*x), name=\
"mrcnn_bbox_loss")(
    [target_bbox, target_class_ids, mrcnn_bbox])
mask_loss = KL.Lambda(lambda x: mrcnn_mask_loss_graph(*x), name=\
"mrcnn_mask_loss")(
    [target_mask, target_class_ids, mrcnn_mask])

# 模型
inputs = [input_image, input_image_meta,
          input_rpn_match, input_rpn_bbox, input_gt_class_ids, input_\
gt_boxes, input_gt_masks]
if not config.USE_RPN_ROIS:
    inputs.append(input_rois)
outputs = [rpn_class_logits, rpn_class, rpn_bbox, mrcnn_class_logits,
mrcnn_class, mrcnn_bbox, mrcnn_mask, rpn_rois, output_rois, rpn_class_\
loss, rpn_bbox_loss, class_loss, bbox_loss, mask_loss]
model = KM.Model(inputs, outputs, name='mask_rcnn')
else:
    mrcnn_class_logits, mrcnn_class, mrcnn_bbox = \
        fpn_classifier_graph(rpn_rois, mrcnn_feature_maps, config.IMAGE_\
SHAPE, config.POOL_SIZE, config.NUM_CLASSES)

    detections = DetectionLayer(config, name="mrcnn_detection")(
        [rpn_rois, mrcnn_class, mrcnn_bbox, input_image_meta]) # 检测

h, w = config.IMAGE_SHAPE[:2] # 将边框转换到归一化坐标系中
detection_boxes = KL.Lambda(lambda x: x[..., :4] / np.array([h, w,
h, w]))(detections)

```

```

# 创建检测掩膜
mrcnn_mask = build_fpn_mask_graph(detection_boxes, mrcnn_feature_maps,
                                   config.IMAGE_SHAPE, config.MASK\
                                   POOL_SIZE,
                                   config.NUM_CLASSES)

model = KM.Model([input_image, input_image_meta], [detections, mrcnn\
class, mrcnn_bbox,
                                   mrcnn_mask, rpn_rois, rpn_class, rpn_bbox], name=\
                                   'mask_rcnn')

if config.GPU_COUNT > 1: # 添加多 GPU 支持
    from parallel_model import ParallelModel
    model = ParallelModel(model, config.GPU_COUNT)
return model

def find_last(self):
    # 在模型目录中找到训练模型的最后 checkpoint 文件
    dir_names = next(os.walk(self.model_dir))[1]
    key = self.config.NAME.lower()
    dir_names = filter(lambda f: f.startswith(key), dir_names)
    dir_names = sorted(dir_names)
    if not dir_names:
        return None, None
    dir_name = os.path.join(self.model_dir, dir_names[-1]) # 选择最后的目录
    checkpoints = next(os.walk(dir_name))[2] # 找到最后的 checkpoint
    checkpoints = filter(lambda f: f.startswith("mask_rcnn"), checkpoints)
    checkpoints = sorted(checkpoints)
    if not checkpoints:
        return dir_name, None
    checkpoint = os.path.join(dir_name, checkpoints[-1])
    return dir_name, checkpoint

def load_weights(self, filepath, by_name=False, exclude=None):
    import h5py
    from keras.engine import topology

    if exclude:
        by_name = True

    if h5py is None:
        raise ImportError("'load_weights' requires h5py.")
    f = h5py.File(filepath, mode='r')
    if 'layer_names' not in f.attrs and 'model_weights' in f:
        f = f['model_weights']

    keras_model = self.keras_model
    layers = keras_model.inner_model.layers if hasattr(keras_model, "inner_\
model")
    else keras_model.layers

    if exclude:
        # 排除一些层
        layers = filter(lambda l: l.name not in exclude, layers)

```



```

    if by_name:
        topology.load_weights_from_hdf5_group_by_name(f, layers)
    else:
        topology.load_weights_from_hdf5_group(f, layers)
    if hasattr(f, 'close'):
        f.close()

    self.set_log_dir(filepath)      # 更新日志目录

def get_imagenet_weights(self):      # 下载用 ImageNet 训练的权值
    from keras.utils.data_utils import get_file
    TF_WEIGHTS_PATH_NO_TOP = 'https://github.com/fchollet/deep-learning-models/'\
        'releases/download/v0.2/'\
        'resnet50_weights_tf_dim_ordering_tf_kernels_\
        notop.h5'
    weights_path = get_file('resnet50_weights_tf_dim_ordering_tf_kernels_\
        notop.h5',
                            TF_WEIGHTS_PATH_NO_TOP, cache_subdir='models',
                            md5_hash='a268eb855778b3df3c7506639542a6af')

    return weights_path

def compile(self, learning_rate, momentum):
    optimizer = keras.optimizers.SGD(lr=learning_rate, momentum=momentum,
        clipnorm=5.0)
    self.keras_model._losses = []
    self.keras_model._per_input_losses = {}
    loss_names = ["rpn_class_loss", "rpn_bbox_loss",
        "mrcnn_class_loss", "mrcnn_bbox_loss", "mrcnn_mask_loss"]
    for name in loss_names:
        layer = self.keras_model.get_layer(name)
        if layer.output in self.keras_model.losses:
            continue
        self.keras_model.add_loss(tf.reduce_mean(layer.output, keep_dims=\
            True))      # 增加损失

    # 添加 L2 正则化
    reg_losses = [keras.regularizers.l2(self.config.WEIGHT_DECAY)(w) / tf.cast\
        (tf.size(w), tf.float32)
        for w in self.keras_model.trainable_weights
        if 'gamma' not in w.name and 'beta' not in w.name]
    self.keras_model.add_loss(tf.add_n(reg_losses))

    self.keras_model.compile(optimizer=optimizer, loss=[None] * len(self.\
        keras_model.outputs))      # 编译

    for name in loss_names:      # 为损失添加度量
        if name in self.keras_model.metrics_names:
            continue
        layer = self.keras_model.get_layer(name)
        self.keras_model.metrics_names.append(name)
        self.keras_model.metrics_tensors.append(tf.reduce_mean(layer.output,
            keep_dims=True))

```

```

def set_trainable(self, layer_regex, keras_model=None, indent=0, verbose=1):
    # 如果模型层的名字匹配给定的正则表达式，那么将模型的层设置为可训练的
    if verbose > 0 and keras_model is None:
        log("Selecting layers to train")

    keras_model = keras_model or self.keras_model
    layers = keras_model.inner_model.layers if hasattr(keras_model, "inner_\
model")\
    else keras_model.layers

    for layer in layers:
        if layer.__class__.__name__ == 'Model':
            print("In model: ", layer.name)
            self.set_trainable(layer_regex, keras_model=layer,
                                indent=indent + 4)
            continue

        if not layer.weights:
            continue
        trainable = bool(re.fullmatch(layer_regex, layer.name))
        if layer.__class__.__name__ == 'TimeDistributed':
            layer.layer.trainable = trainable
        else:
            layer.trainable = trainable
        if trainable and verbose > 0:
            log("{}{:20}   {}".format(" " * indent, layer.name, layer.__\
class__.__name__))

def set_log_dir(self, model_path=None):
    self.epoch = 0
    now = datetime.datetime.now()

    if model_path:
        regex = r".*/w+(\d{4})(\d{2})(\d{2})T(\d{2})(\d{2})/mask\_rcnn\_\\
w+(\d{4})\.h5"
        m = re.match(regex, model_path)
        if m:
            now = datetime.datetime(int(m.group(1)), int(m.group(2)), int(\
m.group(3)),
                                    int(m.group(4)), int(m.group(5)))
            self.epoch = int(m.group(6)) + 1
    self.log_dir = os.path.join(self.model_dir, "{}{:Y%m%dT%H%M}".format(\
self.config.NAME.lower(), now)) # 训练日志的目录

    self.checkpoint_path = os.path.join(self.log_dir, "mask_rcnn_{\_epoch*.\
h5".format(
        self.config.NAME.lower()))
    self.checkpoint_path = self.checkpoint_path.replace("*epoch*", "{epoch:04d}")

def train(self, train_dataset, val_dataset, learning_rate, epochs, layers):
    # 训练模型

```

```
assert self.mode == "training", "Create model in training mode."
layer_regex = {
    "heads": r"(mrcnn\_.*)|(rpn\_.*)|(fpn\_.*)",
    "3+": r"(res3.*)|(bn3.*)|(res4.*)|(bn4.*)|(res5.*)|(bn5.)|(mrcnn\_.*)|(rpn\_.*)|(fpn\_.*)",
    "4+": r"(res4.*)|(bn4.*)|(res5.*)|(bn5.)|(mrcnn\_.*)|(rpn\_.*)|(f\n\_.*)",
    "5+": r"(res5.*)|(bn5.)|(mrcnn\_.*)|(rpn\_.*)|(fpn\_.*)",
    "all": ".*",
}
if layers in layer_regex.keys():
    layers = layer_regex[layers]

train_generator = data_generator(train_dataset, self.config, shuffle=True,
                                batch_size=self.config.BATCH_SIZE)
# 数据生成器
val_generator = data_generator(val_dataset, self.config, shuffle=True,
                              batch_size=self.config.BATCH_SIZE, augment=False)
callbacks = [keras.callbacks.TensorBoard(log_dir=self.log_dir,
                                         histogram_freq=0, write_graph=True, write_images=False),
            keras.callbacks.ModelCheckpoint(self.checkpoint_path, verbose=0, save_weights_only=True), ]

log("\nStarting at epoch {}. LR={}\n".format(self.epoch, learning_rate))
# 训练
log("Checkpoint Path: {}".format(self.checkpoint_path))
self.set_trainable(layers)
self.compile(learning_rate, self.config.LEARNING_MOMENTUM)

if os.name is 'nt':
    workers = 0
else:
    workers = max(self.config.BATCH_SIZE // 2, 2)

self.keras_model.fit_generator(train_generator, initial_epoch=self.\
epoch, epochs=epochs,\
                               steps_per_epoch=self.config.STEPS_PER_EPOCH, callbacks=callbacks,\
                               validation_data=next(val_generator), validation_steps=self.config.VALIDATION_STEPS,\
                               max_queue_size=100, workers=workers, use_multiprocessing=True,)
self.epoch = max(self.epoch, epochs)
```

```
def mold_inputs(self, images): # 获取图像列表，并将它们修改为神经网络要求的输入格式
    molded_images = []
    image metas = []
    windows = []
    for image in images:
```



```

        padding=self.config.IMAGE_PADDING)
    molded_image = mold_image(molded_image, self.config)
    image_meta = compose_image_meta(0, image.shape, window,
        np.zeros([self.config.NUM_CLASSES], dtype=np.int32))
    molded_images.append(molded_image)
    windows.append(window)
    image metas.append(image_meta)
    molded_images = np.stack(molded_images)
    image metas = np.stack(image metas)
    windows = np.stack(windows)
    return molded_images, image metas, windows

# 将神经网络输出的格式转化为其他应用需要的格式
def unmold_detections(self, detections, mrcnn_mask, image_shape, window):
    zero_ix = np.where(detections[:, 4] == 0)[0]
    N = zero_ix[0] if zero_ix.shape[0] > 0 else detections.shape[0]

    boxes = detections[:N, :4]
    class_ids = detections[:N, 4].astype(np.int32)
    scores = detections[:N, 5]
    masks = mrcnn_mask[np.arange(N), :, :, class_ids]

    h_scale = image_shape[0] / (window[2] - window[0])
    w_scale = image_shape[1] / (window[3] - window[1])
    scale = min(h_scale, w_scale)
    shift = window[:2]
    scales = np.array([scale, scale, scale, scale])
    shifts = np.array([shift[0], shift[1], shift[0], shift[1]])
    boxes = np.multiply(boxes - shifts, scales).astype(np.int32)

    exclude_ix = np.where((boxes[:, 2] - boxes[:, 0]) * (boxes[:, 3] - boxes\
[:, 1]) <= 0)[0]
    if exclude_ix.shape[0] > 0:
        boxes = np.delete(boxes, exclude_ix, axis=0)
        class_ids = np.delete(class_ids, exclude_ix, axis=0)
        scores = np.delete(scores, exclude_ix, axis=0)
        masks = np.delete(masks, exclude_ix, axis=0)
        N = class_ids.shape[0]

    full_masks = []
    for i in range(N):
        full_mask = utils.unmold_mask(masks[i], boxes[i], image_shape)
        full_masks.append(full_mask)
    full_masks = np.stack(full_masks, axis=-1)\
    if full_masks else np.empty((0,) + masks.shape[1:3])
    return boxes, class_ids, scores, full_masks

def detect(self, images, verbose=0): # 运行检测流程
    assert self.mode == "inference", "Create model in inference mode."
    assert len(images) == self.config.BATCH_SIZE, "len(images) must be equal\
to BATCH_SIZE"

```

```

if verbose:
    log("Processing {} images".format(len(images)))
    for image in images:
        log("image", image)
# 修改输入, 使其满足神经网络需要的格式
molded_images, image metas, windows = self.mold_inputs(images)
if verbose:
    log("molded_images", molded_images)
    log("image metas", image metas)

detections, mrcnn_class, mrcnn_bbox, mrcnn_mask, \    # 运行目标检测
    rois, rpn_class, rpn_bbox = \
    self.keras_model.predict([molded_images, image metas], verbose=0)

results = []
for i, image in enumerate(images):                # 检测过程
    final_rois, final_class_ids, final_scores, final_masks = \
        self.unmold_detections(detections[i], mrcnn_mask[i], image.shape,
                                windows[i])
    results.append(("rois": final_rois, "class_ids": final_class_ids,
                    "scores": final_scores, "masks": final_masks,))
return results

def ancestor(self, tensor, name, checked=None):
    checked = checked if checked is not None else []
    if len(checked) > 500:
        return None
    if isinstance(name, str):
        name = re.compile(name.replace("/", r"(\_\d+)*\/"))
    parents = tensor.op.inputs
    for p in parents:
        if p in checked:
            continue
        if bool(re.fullmatch(name, p.name)):
            return p
        checked.append(p)
        a = self.ancestor(p, name, checked)
        if a is not None:
            return a
    return None

def find_trainable_layer(self, layer):
    if layer.__class__.__name__ == 'TimeDistributed':
        return self.find_trainable_layer(layer.layer)
    return layer

def get_trainable_layers(self):
    layers = []
    for l in self.keras_model.layers:                # 在所有层中循环
        l = self.find_trainable_layer(l)
        if l.get_weights():

```

```

        layers.append(l)
    return layers

def run_graph(self, images, outputs):
    model = self.keras_model
    outputs = OrderedDict(outputs)
    for o in outputs.values():
        assert o is not None

    inputs = model.inputs
    if model.uses_learning_phase and not isinstance(K.learning_phase(), int):
        inputs += [K.learning_phase()]
    kf = K.function(model.inputs, list(outputs.values()))

    molded_images, image_metas, windows = self.mold_inputs(images)
    model_in = [molded_images, image_metas]
    if model.uses_learning_phase and not isinstance(K.learning_phase(), int):
        model_in.append(0.)
    outputs_np = kf(model_in)

    # 将生成的 Numpy 数组转换为一个字典，并且记录结果
    outputs_np = OrderedDict([(k, v) for k, v in zip(outputs.keys(), outputs_np)])
    for k, v in outputs_np.items():
        log(k, v)
    return outputs_np

```

9.3.3 Mask R-CNN 的图像实例分割案例及演示效果

本节描述一个利用 Mask R-CNN 在 Keras 接口和 TensorFlow 框架下进行图像实例分割的案例，其中用到的 COCO 2014 数据集可以根据表 1.2 提供的地址下载，存放到 ./Mask_RCNN/coco 文件夹下。由于这个数据集测试集没有真实标签，所以本案例采用训练集进行训练，采用验证集进行测试。另外，本案例还需要从 https://github.com/matterport/Mask_RCNN/releases 和 <https://github.com/philferriere/cocoapi> 分别下载初始化权值和接口文件，解压后存放到 Mask_RCNN 文件夹中，并进入 ./Mask_RCNN/cocoapi/PythonAPI/ 目录，分别按照图 9.20 和图 9.21 中的命令，进行编译和安装此包。注意，必须使用 Visual Studio 2015 进行编译，应确保计算机中已安装 Visual Studio 2015，且 Python 安装目录 ./Lib/distutils/ 中文件 msvc9compiler.py 的第 234 行为 “toolskey = “VS140COMNTOOLS” % version”。

Mask R-CNN 的训练可参照图 9.22 的命令执行。训练过程分为 3 个阶段：第 1 阶段训练网络的头部，迭代次数为 40；第 2 阶段训练残差网络 conv4_x 及以上部分，迭代次数为 120；第 3 阶段训练整个网络，迭代次数为 160。图 9.23 ~ 图 9.25 分别给出了这 3 个阶段的训练信息。图 9.26 给出了网络在训练结束时的显示信息。在训练结束之后，即可参照图 9.27 的命令验证实例分割的效果，如图 9.28 所示。

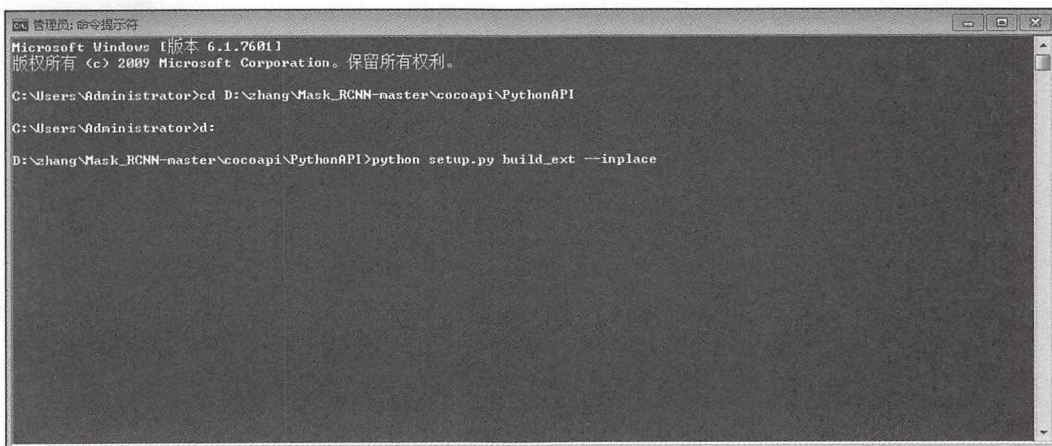


图 9.20 cocoapi 的编译命令

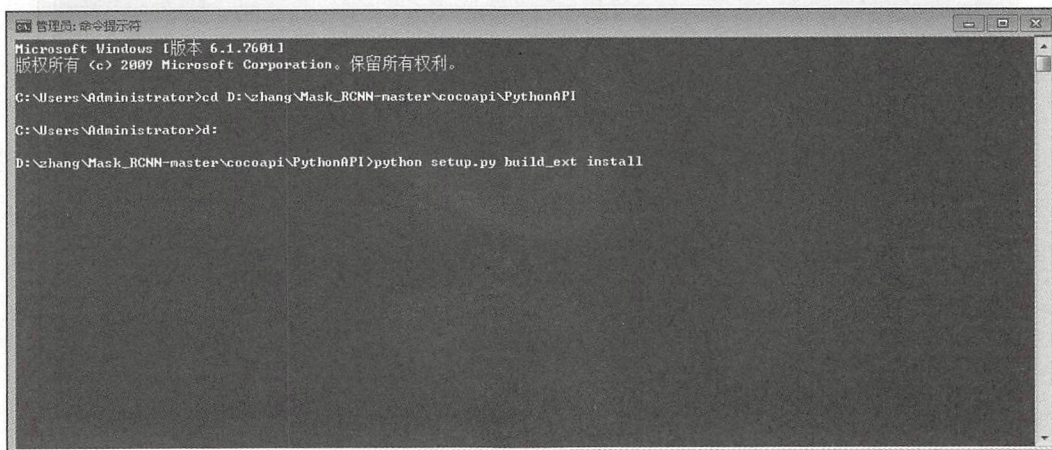


图 9.21 cocoapi 的安装命令

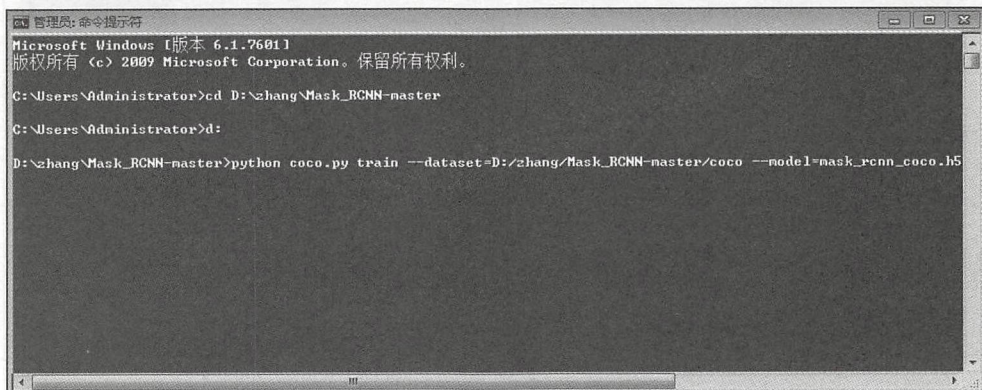


图 9.22 Mask R-CNN 图像实例分割案例的训练命令


```

管理提示符: python coco.py train --dataset=coco --model=mask_rcnn_coco_h5
985/1000 [.....] ETA: 55s - loss: 0.8314 - rpn_class_loss: 0.0143 - rpn_bbox_loss: 0.1914 - mrcnn_
986/1000 [.....] ETA: 51s - loss: 0.8309 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1913 - mrcnn_
987/1000 [.....] ETA: 48s - loss: 0.8312 - rpn_class_loss: 0.0143 - rpn_bbox_loss: 0.1913 - mrcnn_
988/1000 [.....] ETA: 44s - loss: 0.8308 - rpn_class_loss: 0.0143 - rpn_bbox_loss: 0.1913 - mrcnn_
989/1000 [.....] ETA: 40s - loss: 0.8304 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1914 - mrcnn_
990/1000 [.....] ETA: 37s - loss: 0.8303 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1913 - mrcnn_
991/1000 [.....] ETA: 33s - loss: 0.8300 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1913 - mrcnn_
992/1000 [.....] ETA: 29s - loss: 0.8304 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1913 - mrcnn_
993/1000 [.....] ETA: 25s - loss: 0.8299 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1912 - mrcnn_
994/1000 [.....] ETA: 22s - loss: 0.8295 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1911 - mrcnn_
995/1000 [.....] ETA: 18s - loss: 0.8296 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1912 - mrcnn_
996/1000 [.....] ETA: 14s - loss: 0.8293 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1912 - mrcnn_
997/1000 [.....] ETA: 11s - loss: 0.8288 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1911 - mrcnn_
998/1000 [.....] ETA: 7s - loss: 0.8288 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1910 - mrcnn_
999/1000 [.....] ETA: 3s - loss: 0.8287 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1910 - mrcnn_
1000/1000 [.....] - 3732s 4s/step - loss: 0.8289 - rpn_class_loss: 0.0142 - rpn_bbox_loss: 0.1909 -
mrcnn_class_loss: 0.1965 mrcnn_bbox_loss: 0.1648 mrcnn_mask_loss: 0.2624 - val_loss: 1.4342 - val_rpn_class_loss: 0.0263
- val_rpn_bbox_loss: 0.4793 - val_mrcnn_class_loss: 0.2340 - val_mrcnn_bbox_loss: 0.3278 - val_mrcnn_mask_loss: 0.3667
Epoch 2/40
1/1000 [.....] ETA: 1:10:30 - loss: 1.1825 - rpn_class_loss: 0.0397 - rpn_bbox_loss: 0.2201 - mrcnn_
2/1000 [.....] ETA: 1:06:41 - loss: 0.8537 - rpn_class_loss: 0.0219 - rpn_bbox_loss: 0.1627 - mrcnn_
3/1000 [.....] ETA: 1:03:58 - loss: 0.8449 - rpn_class_loss: 0.0159 - rpn_bbox_loss: 0.1614 - mrcnn_
4/1000 [.....] ETA: 1:06:56 - loss: 0.8326 - rpn_class_loss: 0.0133 - rpn_bbox_loss: 0.1663 - mrcnn_
5/1000 [.....] ETA: 1:04:43 - loss: 0.8250 - rpn_class_loss: 0.0112 - rpn_bbox_loss: 0.1668 - mrcnn_
mrcnn_class_loss: 0.2143 mrcnn_bbox_loss: 0.1704 mrcnn_mask_loss: 0.2623

```

图 9.23 Mask R-CNN 图像实例分割案例程序在第 1 阶段的训练信息

```

管理提示符: python coco.py train --dataset=coco --model=mask_rcnn_coco_h5
971/1000 [.....] ETA: 2:15 - loss: 0.9131 - rpn_class_loss: 0.0167 - rpn_bbox_loss: 0.2214 - mrcnn_
972/1000 [.....] ETA: 2:10 - loss: 0.9136 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2216 - mrcnn_
973/1000 [.....] ETA: 2:06 - loss: 0.9135 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2215 - mrcnn_
974/1000 [.....] ETA: 2:01 - loss: 0.9131 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2215 - mrcnn_
975/1000 [.....] ETA: 1:56 - loss: 0.9135 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2215 - mrcnn_
976/1000 [.....] ETA: 1:52 - loss: 0.9133 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2216 - mrcnn_
977/1000 [.....] ETA: 1:47 - loss: 0.9128 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2214 - mrcnn_
978/1000 [.....] ETA: 1:42 - loss: 0.9128 - rpn_class_loss: 0.0167 - rpn_bbox_loss: 0.2213 - mrcnn_
979/1000 [.....] ETA: 1:38 - loss: 0.9125 - rpn_class_loss: 0.0167 - rpn_bbox_loss: 0.2212 - mrcnn_
980/1000 [.....] ETA: 1:33 - loss: 0.9126 - rpn_class_loss: 0.0167 - rpn_bbox_loss: 0.2211 - mrcnn_
981/1000 [.....] ETA: 1:28 - loss: 0.9122 - rpn_class_loss: 0.0167 - rpn_bbox_loss: 0.2209 - mrcnn_
982/1000 [.....] ETA: 1:24 - loss: 0.9128 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2212 - mrcnn_
983/1000 [.....] ETA: 1:19 - loss: 0.9126 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2212 - mrcnn_
984/1000 [.....] ETA: 1:14 - loss: 0.9126 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2212 - mrcnn_
985/1000 [.....] ETA: 1:10 - loss: 0.9128 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2211 - mrcnn_
986/1000 [.....] ETA: 1:06 - loss: 0.9128 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2210 - mrcnn_
987/1000 [.....] ETA: 1:00 - loss: 0.9121 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2211 - mrcnn_
988/1000 [.....] ETA: 56s - loss: 0.9122 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2211 - mrcnn_
989/1000 [.....] ETA: 51s - loss: 0.9124 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2210 - mrcnn_
990/1000 [.....] ETA: 46s - loss: 0.9125 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2209 - mrcnn_
991/1000 [.....] ETA: 42s - loss: 0.9128 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2210 - mrcnn_
992/1000 [.....] ETA: 37s - loss: 0.9126 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2209 - mrcnn_
993/1000 [.....] ETA: 32s - loss: 0.9121 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2208 - mrcnn_
994/1000 [.....] ETA: 28s - loss: 0.9127 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2209 - mrcnn_
995/1000 [.....] ETA: 23s - loss: 0.9123 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2208 - mrcnn_
996/1000 [.....] ETA: 18s - loss: 0.9127 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2212 - mrcnn_
997/1000 [.....] ETA: 14s - loss: 0.9131 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2213 - mrcnn_
998/1000 [.....] ETA: 9s - loss: 0.9131 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2212 - mrcnn_
999/1000 [.....] ETA: 4s - loss: 0.9133 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2212 - mrcnn_
1000/1000 [.....] - 4681s 5s/step - loss: 0.9133 - rpn_class_loss: 0.0168 - rpn_bbox_loss: 0.2211 - mrcnn_
mrcnn_class_loss: 0.1803 mrcnn_mask_loss: 0.2651 - val_loss: 0.9547 - val_rpn_class_loss: 0.0018 - val_rpn_bbox_loss: 0.2065 - val_mrcnn_
bbox_loss: 0.2576 - val_mrcnn_mask_loss: 0.3736
Epoch 29/120
1/1000 [.....] ETA: 1:10:42 - loss: 1.1759 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.2221 - mrcnn_
2/1000 [.....] ETA: 1:20:58 - loss: 1.1851 - rpn_class_loss: 0.0318 - rpn_bbox_loss: 0.3891 - mrcnn_
- loss: 0.2431 - mrcnn_mask_loss: 0.3162

```

图 9.24 Mask R-CNN 图像实例分割案例程序第 2 阶段的训练信息

```

管理提示符: python coco.py train --dataset=coco --model=mask_rcnn_coco_h5
972/1000 [.....] ETA: 2:17 - loss: 0.8438 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1960 - mrcnn_
973/1000 [.....] ETA: 2:12 - loss: 0.8438 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1960 - mrcnn_
974/1000 [.....] ETA: 2:07 - loss: 0.8439 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1960 - mrcnn_
975/1000 [.....] ETA: 2:02 - loss: 0.8439 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1959 - mrcnn_
976/1000 [.....] ETA: 1:57 - loss: 0.8435 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1958 - mrcnn_
977/1000 [.....] ETA: 1:52 - loss: 0.8438 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1959 - mrcnn_
978/1000 [.....] ETA: 1:47 - loss: 0.8436 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1958 - mrcnn_
979/1000 [.....] ETA: 1:43 - loss: 0.8438 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1958 - mrcnn_
980/1000 [.....] ETA: 1:39 - loss: 0.8440 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1958 - mrcnn_
981/1000 [.....] ETA: 1:33 - loss: 0.8440 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1958 - mrcnn_
982/1000 [.....] ETA: 1:28 - loss: 0.8445 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1959 - mrcnn_
983/1000 [.....] ETA: 1:23 - loss: 0.8445 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1963 - mrcnn_
984/1000 [.....] ETA: 1:18 - loss: 0.8445 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1963 - mrcnn_
985/1000 [.....] ETA: 1:13 - loss: 0.8444 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1962 - mrcnn_
986/1000 [.....] ETA: 1:08 - loss: 0.8439 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1961 - mrcnn_
987/1000 [.....] ETA: 1:03 - loss: 0.8444 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1966 - mrcnn_
988/1000 [.....] ETA: 58s - loss: 0.8438 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1965 - mrcnn_
989/1000 [.....] ETA: 54s - loss: 0.8438 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1964 - mrcnn_
990/1000 [.....] ETA: 49s - loss: 0.8439 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1964 - mrcnn_
991/1000 [.....] ETA: 44s - loss: 0.8442 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1965 - mrcnn_
992/1000 [.....] ETA: 39s - loss: 0.8439 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1964 - mrcnn_
993/1000 [.....] ETA: 34s - loss: 0.8439 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1964 - mrcnn_
994/1000 [.....] ETA: 29s - loss: 0.8436 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1963 - mrcnn_
995/1000 [.....] ETA: 24s - loss: 0.8436 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1962 - mrcnn_
996/1000 [.....] ETA: 19s - loss: 0.8440 - rpn_class_loss: 0.0150 - rpn_bbox_loss: 0.1964 - mrcnn_
997/1000 [.....] ETA: 14s - loss: 0.8432 - rpn_class_loss: 0.0149 - rpn_bbox_loss: 0.1962 - mrcnn_
998/1000 [.....] ETA: 9s - loss: 0.8431 - rpn_class_loss: 0.0149 - rpn_bbox_loss: 0.1962 - mrcnn_
999/1000 [.....] ETA: 4s - loss: 0.8428 - rpn_class_loss: 0.0149 - rpn_bbox_loss: 0.1961 - mrcnn_
1000/1000 [.....] - 4914s 5s/step - loss: 0.8430 - rpn_class_loss: 0.0149 - rpn_bbox_loss: 0.1962 - mrcnn_
mrcnn_class_loss: 0.1669 mrcnn_mask_loss: 0.2623 - val_loss: 1.2744 - val_rpn_class_loss: 0.0119 - val_rpn_bbox_loss: 0.2223 - val_mrcnn_
bbox_loss: 0.2700 - val_mrcnn_mask_loss: 0.3195
Epoch 123/160
1/1000 [.....] ETA: 1:17:10 - loss: 0.3091 - rpn_class_loss: 0.0012 - rpn_bbox_loss: 0.0503 - mrcnn_
2/1000 [.....] ETA: 1:26:10 - loss: 0.5525 - rpn_class_loss: 0.0078 - rpn_bbox_loss: 0.0812 - mrcnn_
3/1000 [.....] ETA: 1:22:26 - loss: 0.5510 - rpn_class_loss: 0.0115 - rpn_bbox_loss: 0.0920 - mrcnn_
- loss: 0.1189 - mrcnn_mask_loss: 0.1072

```

图 9.25 Mask R-CNN 图像实例分割案例程序第 3 阶段的训练信息


```

C:\Users\Administrator>cmd
D:\zhang\Mask_RCNN-master>
967/1000 [.....] - EIoU: 2132 - loss: 0.7774 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1823 - mrcnn_class_loss: 0.1728 - mrcnn_bbox_
970/1000 [.....] - EIoU: 2127 - loss: 0.7774 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1822 - mrcnn_class_loss: 0.1728 - mrcnn_bbox_
972/1000 [.....] - EIoU: 2122 - loss: 0.7768 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1821 - mrcnn_class_loss: 0.1726 - mrcnn_bbox_
973/1000 [.....] - EIoU: 2117 - loss: 0.7766 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1820 - mrcnn_class_loss: 0.1725 - mrcnn_bbox_
974/1000 [.....] - EIoU: 2112 - loss: 0.7763 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1819 - mrcnn_class_loss: 0.1724 - mrcnn_bbox_
975/1000 [.....] - EIoU: 2107 - loss: 0.7757 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1818 - mrcnn_class_loss: 0.1722 - mrcnn_bbox_
976/1000 [.....] - EIoU: 2102 - loss: 0.7752 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1817 - mrcnn_class_loss: 0.1720 - mrcnn_bbox_
977/1000 [.....] - EIoU: 2102 - loss: 0.7752 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1816 - mrcnn_class_loss: 0.1720 - mrcnn_bbox_
978/1000 [.....] - EIoU: 2102 - loss: 0.7752 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1815 - mrcnn_class_loss: 0.1719 - mrcnn_bbox_
979/1000 [.....] - EIoU: 2102 - loss: 0.7752 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1815 - mrcnn_class_loss: 0.1719 - mrcnn_bbox_
980/1000 [.....] - EIoU: 2102 - loss: 0.7752 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1818 - mrcnn_class_loss: 0.1719 - mrcnn_bbox_
981/1000 [.....] - EIoU: 2102 - loss: 0.7752 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1818 - mrcnn_class_loss: 0.1718 - mrcnn_bbox_
982/1000 [.....] - EIoU: 2102 - loss: 0.7752 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1817 - mrcnn_class_loss: 0.1718 - mrcnn_bbox_
983/1000 [.....] - EIoU: 2102 - loss: 0.7752 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1816 - mrcnn_class_loss: 0.1718 - mrcnn_bbox_
984/1000 [.....] - EIoU: 2102 - loss: 0.7747 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1815 - mrcnn_class_loss: 0.1719 - mrcnn_bbox_
985/1000 [.....] - EIoU: 2102 - loss: 0.7749 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1815 - mrcnn_class_loss: 0.1721 - mrcnn_bbox_
986/1000 [.....] - EIoU: 2102 - loss: 0.7748 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1814 - mrcnn_class_loss: 0.1721 - mrcnn_bbox_
987/1000 [.....] - EIoU: 2102 - loss: 0.7744 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1813 - mrcnn_class_loss: 0.1720 - mrcnn_bbox_
988/1000 [.....] - EIoU: 2102 - loss: 0.7737 - rpn_class_loss: 0.0129 - rpn_bbox_loss: 0.1811 - mrcnn_class_loss: 0.1718 - mrcnn_bbox_
989/1000 [.....] - EIoU: 2102 - loss: 0.7736 - rpn_class_loss: 0.0129 - rpn_bbox_loss: 0.1811 - mrcnn_class_loss: 0.1718 - mrcnn_bbox_
990/1000 [.....] - EIoU: 2102 - loss: 0.7735 - rpn_class_loss: 0.0129 - rpn_bbox_loss: 0.1811 - mrcnn_class_loss: 0.1718 - mrcnn_bbox_
991/1000 [.....] - EIoU: 2102 - loss: 0.7739 - rpn_class_loss: 0.0129 - rpn_bbox_loss: 0.1810 - mrcnn_class_loss: 0.1719 - mrcnn_bbox_
992/1000 [.....] - EIoU: 2102 - loss: 0.7734 - rpn_class_loss: 0.0129 - rpn_bbox_loss: 0.1810 - mrcnn_class_loss: 0.1717 - mrcnn_bbox_
993/1000 [.....] - EIoU: 2102 - loss: 0.7736 - rpn_class_loss: 0.0129 - rpn_bbox_loss: 0.1810 - mrcnn_class_loss: 0.1718 - mrcnn_bbox_
994/1000 [.....] - EIoU: 2102 - loss: 0.7735 - rpn_class_loss: 0.0129 - rpn_bbox_loss: 0.1809 - mrcnn_class_loss: 0.1718 - mrcnn_bbox_
995/1000 [.....] - EIoU: 2102 - loss: 0.7740 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1809 - mrcnn_class_loss: 0.1721 - mrcnn_bbox_
996/1000 [.....] - EIoU: 2102 - loss: 0.7741 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1809 - mrcnn_class_loss: 0.1720 - mrcnn_bbox_
997/1000 [.....] - EIoU: 2102 - loss: 0.7740 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1809 - mrcnn_class_loss: 0.1720 - mrcnn_bbox_
998/1000 [.....] - EIoU: 2102 - loss: 0.7747 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1815 - mrcnn_class_loss: 0.1728 - mrcnn_bbox_
999/1000 [.....] - EIoU: 2102 - loss: 0.7741 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1814 - mrcnn_class_loss: 0.1719 - mrcnn_bbox_
1000/1000 [.....] - EIoU: 56/step - loss: 0.7737 - rpn_class_loss: 0.0130 - rpn_bbox_loss: 0.1813 - mrcnn_class_loss: 0.1718 - mrcnn_b
ox_loss: 0.1536 - mrcnn_mask_loss: 0.2541 - val_loss: 1.3606 - val_rpn_class_loss: 0.0389 - val_rpn_bbox_loss: 0.2272 - val_mrcnn_class_loss: 0.4788 - val_mrcnn
bbox_loss: 0.3075 - val_mrcnn_mask_loss: 0.3077
D:\zhang\Mask_RCNN-master>

```

图 9.26 Mask R-CNN 图像实例分割案例程序在训练结束时的显示信息

```

C:\Users\Administrator>cd D:\zhang\Mask_RCNN-master
C:\Users\Administrator>cd
D:\zhang\Mask_RCNN-master>python coco.py evaluate --dataset=D:\zhang\Mask_RCNN-master\coco --model=D:\zhang\Mask_RCNN-master\
logs\coco20180119T1540\nask_rcnn_coco_0160.h5

```

图 9.27 Mask R-CNN 图像实例分割案例程序的验证命令

```

Done (t=0.86s)
creating index...
index created!
Running COCO evaluation on 500 images.
Loading and preparing results...
DONE (t=0.01s)
creating index...
index created!
Running per image evaluation...
Evaluate annotation type *bbox*
DONE (t=1.83s).
Accumulating evaluation results...
DONE (t=0.67s).
Average Precision (AP) @ IoU=0.50:0.95 : area= all : maxDets=100 : 0.369
Average Precision (AP) @ IoU=0.50 : area= all : maxDets=100 : 0.559
Average Precision (AP) @ IoU=0.75 : area= all : maxDets=100 : 0.409
Average Precision (AP) @ IoU=0.50:0.95 : area= small : maxDets=100 : 0.190
Average Precision (AP) @ IoU=0.50:0.95 : area= medium : maxDets=100 : 0.424
Average Precision (AP) @ IoU=0.50:0.95 : area= large : maxDets=100 : 0.549
Average Recall (AR) @ IoU=0.50:0.95 : area= all : maxDets=1 : 0.300
Average Recall (AR) @ IoU=0.50:0.95 : area= all : maxDets=10 : 0.425
Average Recall (AR) @ IoU=0.50:0.95 : area= all : maxDets=100 : 0.437
Average Recall (AR) @ IoU=0.50:0.95 : area= small : maxDets=100 : 0.226
Average Recall (AR) @ IoU=0.50:0.95 : area= medium : maxDets=100 : 0.487
Average Recall (AR) @ IoU=0.50:0.95 : area= large : maxDets=100 : 0.624
Prediction time: 469.54472494125366. Average 0.9390894498825073/image
Total time: 485.25996494293213

```

图 9.28 Mask R-CNN 图像实例分割案例程序的验证结果

从图 9.23 中可以看出，网络在第 1 阶段训练到 1 次时，总训练损失为 0.8289，RPN 训练的分类损失和边框回归损失分别为 0.0142 和 0.1909，Mask R-CNN 训练的分类损失、边框回归损失和掩膜损失分别为 0.1965、0.1648 和 0.2624；总验证损失为 1.4342，RPN 验证的分类损失和边框回归损失分别为 0.0263 和 0.4793，Mask R-CNN 验证的分类损失、边框回归损失和掩膜损失分别为 0.2340、0.3278 和 0.3667。

从图 9.24 中可以看出，网络在第 2 阶段训练到 98 次时，总训练损失为 0.9133，RPN 训练的分类损失和边框回归损失分别为 0.0168 和 0.2211，Mask R-CNN 训练的分类损失、边框回归损失和掩膜损失分别为 0.2299、0.1803 和 0.2651；总验证损失为 0.9547，RPN 验证的分类损失和边框回归损失分别为 0.0018 和 0.2065，Mask R-CNN 验证的分类损失、边框回归损失和掩膜损失分别为 0.1153、0.2576 和 0.3736。

从图 9.25 中可以看出，网络在第 3 阶段训练到 122 次时，总训练损失为 0.8430，RPN 训练的分类损失和边框回归损失分别为 0.0149 和 0.1962，Mask R-CNN 训练的分类损失、边框回归损失和掩膜损失分别为 0.2028、0.1669 和 0.2623；总验证损失为 1.2924，RPN 验证的分类损失和边框回归损失分别为 0.0419 和 0.2223，Mask R-CNN 验证的分类损失、边框回归损失和掩膜损失分别为 0.4377、0.2700 和 0.3195。

从图 9.26 中可以看出，网络在训练结束时，总训练损失为 0.7737，RPN 训练的分类损失和边框回归损失分别为 0.0130 和 0.1813，Mask R-CNN 训练的分类损失、边框回归损失和掩膜损失分别为 0.1718、0.1536 和 0.2541；总验证损失为 0.0389，RPN 验证的分类损失和边框回归损失分别为 0.0389 和 0.2277，Mask R-CNN 验证的分类损失、边框回归损失和掩膜损失分别为 0.4788、0.3075 和 0.3077。

Mask R-CNN 图像实例分割案例程序的验证命令如图 9.27 所示，执行此命令，结果如图 9.28 所示。从图 9.28 可以看出，图像实例分割的好坏是采用平均准确率和平均召回率来评价的。平均准确率的计算分 6 种情况：IoU 为 0.5 ~ 0.95 的平均准确率为 36.9%，IoU 为 0.5 的平均准确率为 55.9%，IoU 为 0.75 的平均准确率为 40.9%，小物体（区域小于 32×32 的物体）的平均准确率为 19.0%，中等物体（区域为 32×32 到 96×96 之间的物体）的平均准确率为 42.4%，大物体（区域大于 96×96 的物体）的平均准确率为 54.9%。平均召回率的计算也有分 6 种情况：每幅图像采用 top-1 分割结果的平均召回率为 30.0%，每幅图像采用 top-10 分割结果的平均召回率为 42.5%，每幅图像采用 top-100 分割结果的平均召回率为 43.7%，小物体（区域小于 32×32 的物体）的平均召回率为 22.6%，中等物体（区域为 32×32 到 96×96 之间的物体）的平均召回率为 48.7%，大物体（区域大于 96×96 的物体）的平均召回率为 62.4%。此外，还可以看到，总的测试时间约为 469s，平均每幅图像约 0.94s，总的运行时间约为 485s。注意，程序默认验证 500 幅图像，以上数据是在 500 幅图像上的验证结果，并非是整个验证集的结果。

最后，利用图 9.29 的命令能够对图像实例分割的结果进行可视化，该命令执行后会产生一个如图 9.30 所示的网页界面，在界面中单击 demo.ipynb 文件，跳转到如图 9.31 所示的

网页界面，单击其中的运行按钮，就能够可视化一幅原始图像的分割结果，如图 9.32 所示。

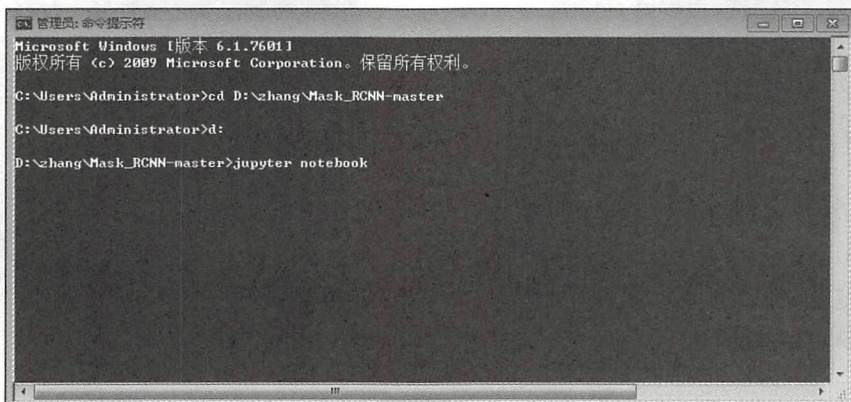


图 9.29 Mask R-CNN 图像实例分割案例程序的可视化命令

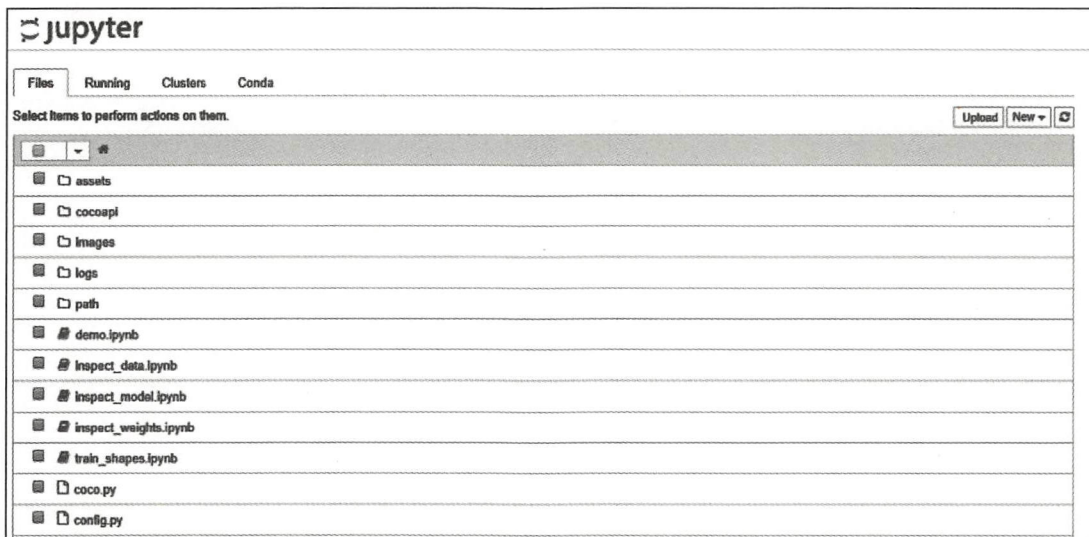


图 9.30 Jupyter Notebook 的编辑界面

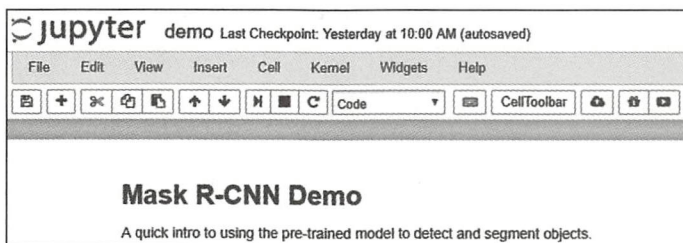


图 9.31 Mask R-CNN 图像实例分割案例程序的可视化页面



a) 原始图像



b) 实例分割结果

图 9.32 Mask R-CNN 图像实例分割案例程序的原始图像和实例分割结果

卷积神经网络的特殊模型

目前, 卷积神经网络的变种模型已经非常丰富, 其中有些结构比较特殊或者用途比较特殊, 暂时不便归类。本书将这些模型统称为特殊模型, 包括孪生网络、挤压网络、生成对抗网络和网中网等。孪生网络具有两个结构和权值完全相同的网络模块, 挤压网络具有更简单的结构和更少的权值, 生成对抗网络具有两个功能互相对抗的网络模块, 网中网采用微网络卷积核代替线性卷积核。本章主要介绍它们的模型结构、Caffe 或 TensorFlow 代码、应用案例及演示效果。

10.1 孪生网络 SiameseNet

10.1.1 SiameseNet 的模型结构

传统的分类模型需要确切知道每个样本的标签属于哪个类, 而标签的数量通常相对较少。在类别数量特别多、标签相对少的情况下, 有些类别可能根本没有标签, 比如人类对第一次见到的生物物种往往是叫不出名字的。这时进行分类就可以考虑采用孪生网络 (Siamese Network, SiameseNet) ^[154]。孪生网络不仅能够从给定数据中学习一个相似性度量, 而且还能够利用所学到的度量去比较和匹配新样本以确定类别。孪生网络的基本思想是构造一个函数将输入映射到目标空间, 在目标空间通过简单的距离 (例如, 欧氏距离) 计算相似度, 总体结构如图 10.1 所示。

孪生网络在本质上是一种判断两个输入模式是否相似的通用框架, 优点在于对领域知识的依赖性较低。从图 10.1 不难看出, 孪

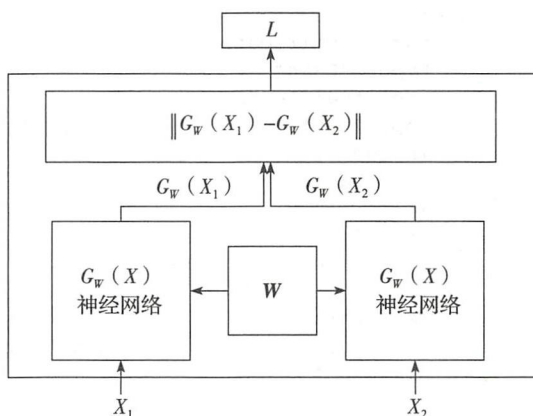


图 10.1 孪生网络的总体结构示意图

生网络有两个输入 X_1 和 X_2 (比如两幅图像), 它们分别同时经过两个结构相同、权值绑定的双胞胎子网络, 得到低维表示 $G_w(X_1)$ 和 $G_w(X_2)$, 最后输出它们之间的相容性, 用 L 表示如下:

$$L = E_w(X_1, X_2) = \|G_w(X_1) - G_w(X_2)\| \quad (10.1)$$

由于双胞胎子网络计算的是相同的函数, 所以能够保证两个非常相似的输入不会被各自的子网络映射到目标空间中非常不同的位置, 从而近似维持它们在输入空间的“语义”距离。此外, 如果两个输入不太相似, 属于不同的类别, 双胞胎子网络应该输出两个差别较大的结果。图 10.2 给出了一个具体的孪生网络结构, 可以用来判断两个 MNIST 手写数字是否为同类, 其中的子网络都包括 2 个交错的卷积层和池化层、3 个全连接层。

孪生网络已经在人脸验证和识别中获得了非常成功的应用^[155-157], 但注意在进行识别和分类时, 孪生网络通常还需要与其他的模型联合使用。

10.1.2 SiameseNet 的 Caffe 代码实现及说明

关于孪生网络的代码, 在 Windows 环境的 Caffe 框架中有一个例子, 位于文件夹 `./Caffe/examples/siamese` 下, 包含网络结构定义文件 `mnist_siamese_train_test.prototxt` 和求解器配置文件 `mnist_siamese_solver.prototxt`。下面参考图 10.2 的具体孪生网络结构, 对其中的几个关键部分进行说明, 包括数据层 `pair_data`、切片层 `slice_pair`、卷积层 `conv1`、池化层 `pool1`、全连接层 `ip1` 和损失层 `loss`。

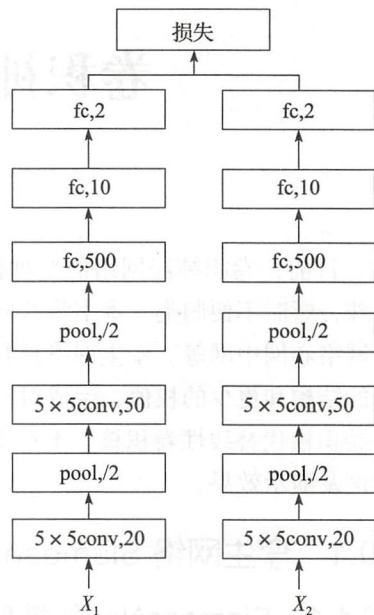


图 10.2 一个具体的孪生网络结构

1. 数据层 `pair_data` 的实现代码及说明

```

layer {
  name: "pair_data"           # 表示该层为数据层
  type: "Data"
  top: "pair_data"
  top: "sim"                  # 表示该图像的标签
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "F:/caffe-windows-master/caffe-windows-master/examples/siamese/trainleveldb"
    batch_size: 64
  }
}

```

2. 切片层 slice_pair 的实现代码及说明

```
layer {
  name: "slice_pair"
  type: "Slice"
  bottom: "pair_data"
  top: "data"
  top: "data_p"
  slice_param {
    slice_dim: 1
    slice_point: 1
  }
}
```

表示该层为切片层
底层为图像对
高层为两幅单独的图像
表示目标维度
表示选定维度的索引

3. 卷积层 conv1 的实现代码及说明

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    name: "conv1_w"
    lr_mult: 1
  }
  param {
    name: "conv1_b"
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

表示该层为卷积层
输出特征图的个数
卷积核的大小
卷积核的步长

4. 池化层 pool1 的实现代码及说明

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
  }
}
```

表示该层为池化层
池化窗口的大小


```

        stride: 2                                # 池化窗口的步长
    }
}

```

5. 全连接层 ip1 的实现代码及说明

```

layer {
    name: "ip1"
    type: "InnerProduct"                        # 表示该层为全连接层
    bottom: "pool2"
    top: "ip1"
    param {
        name: "ip1_w"
        lr_mult: 1
    }
    param {
        name: "ip1_b"
        lr_mult: 2
    }
    inner_product_param {
        num_output: 500                        # 全连接层的节点数
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
layer {
    name: "relu1"
    type: "ReLU"
    bottom: "ip1"
    top: "ip1"
}

```

6. 损失层 loss 的实现代码及说明

```

layer {
    name: "loss"
    type: "ContrastiveLoss"
    bottom: "feat"                             # 第一个图像的低维表示
    bottom: "feat_p"                           # 第二个图像的低维表示
    bottom: "sim"                              # 表示标签
    top: "loss"
    contrastive_loss_param {
        margin: 1
    }
}

```

10.1.3 SiameseNet 的手写数字验证案例

本节描述一个利用 SiameseNet 在 Caffe 框架下进行手写数字验证的案例，其中用到的

MNIST 数据集可以根据表 1.2 提供的地址下载。下载后需要进行数据格式转换，具体方法是将其中的 4 个压缩文件存放到文件夹 `./Caffe/Build/x64/Release` 下，并按方框 10.1 编写批处理文件 `datasia.bat`，分别将训练集和测试集转换成 LEVELDB 格式的文件。此外，还需要按照方框 10.2 修改求解器配置文件 `mnist_siamese_solver.prototxt`，并按方框 10.3 修改网络结构文件 `mnist_siamese_train_test.prototxt`。

方框 10.1 批处理文件 `datasia.bat`

```
convert_mnist_siamese_data.exe train-images-idx3-ubyte train-labels-idx1-ubyte
trainleveldb
convert_mnist_siamese_data.exe t10k-images-idx3-ubyte t10k-labels-idx1-ubyte test-
leveldb
```

方框 10.2 在 `mnist_siamese_solver.prototxt` 中设置的超参数情况

```
net: "F:/caffe-windows-master/caffe-windows-master/examples/siamese/mnist_siamese_
train_test.prototxt"
test_iter: 100
test_interval: 500
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0000
lr_policy: "inv"
gamma: 0.0001
power: 0.75
display: 100
max_iter: 50000
snapshot: 5000
snapshot_prefix: "F:/caffe-windows-master/caffe-windows-master/examples/siamese/
mnist_siamese"
solver_mode: CPU
```

方框 10.3 在 `mnist_siamese_train_test.prototxt` 中的修改情况

```
layer {
  name: "pair_data"
  type: "Data"
  top: "pair_data"
  top: "sim"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "F:/caffe-windows-master/caffe-windows-master/examples/siamese/
trainleveldb"
    batch_size: 64
  }
}
```

```
}  
layer {  
  name: "pair_data"  
  type: "Data"  
  top: "pair_data"  
  top: "sim"  
  include {  
    phase: TEST  
  }  
  transform_param {  
    scale: 0.00390625  
  }  
  data_param {  
    source: "F:/caffe-windows-master/caffe-windows-master/examples/siamese/  
testleveldb"  
    batch_size: 100  
  }  
}
```

在完成上述准备之后，即可对 SiameseNet 的手写数字验证案例程序进行训练和测试。训练命令如图 10.3 所示，训练的中间结果如图 10.4 所示，最终结果如图 10.5 所示。如图 10.4 所示，在训练到 1200 次时，学习率为 0.009 185 15，训练损失为 0.000 260 927；在训练到 1300 次时，学习率为 0.009 124 12，训练损失为 0.000 218 446。如图 10.5 所示，在训练到 50 000 次时，训练损失为 1.829 72e-006，测试损失为 1.400 23e-006。

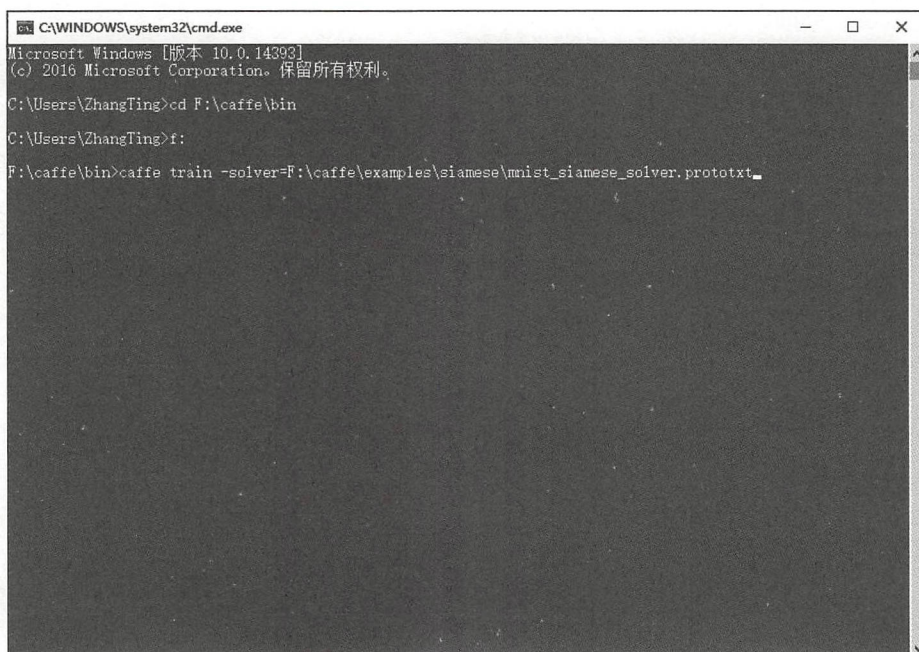


图 10.3 SiameseNet 手写数字验证案例程序的训练命令


```
C:\WINDOWS\system32\cmd.exe - caffe train -solver=F:\caffe\examples\siamese\mnist_siamese_solver.prototxt
I0922 17:04:28.150295 16604 sgd_solver.cpp:106] Iteration 300, lr = 0.00943913
I0922 17:04:37.881585 16604 solver.cpp:228] Iteration 900, loss = 0.000447531
I0922 17:04:37.882586 16604 solver.cpp:244] Train net output #0: loss = 0.000447531 (* 1 = 0.000447531
loss)
I0922 17:04:37.882586 16604 sgd_solver.cpp:106] Iteration 900, lr = 0.00937411
I0922 17:04:47.482071 16604 solver.cpp:337] Iteration 1000, Testing net (#0)
I0922 17:04:54.152848 16604 solver.cpp:404] Test net output #0: loss = 0.000378199 (* 1 = 0.000378199
loss)
I0922 17:04:54.262223 16604 solver.cpp:228] Iteration 1000, loss = 0.000431259
I0922 17:04:54.262223 16604 solver.cpp:244] Train net output #0: loss = 0.000431259 (* 1 = 0.000431259
loss)
I0922 17:04:54.262223 16604 sgd_solver.cpp:106] Iteration 1000, lr = 0.00931012
I0922 17:05:04.050149 16604 solver.cpp:228] Iteration 1100, loss = 0.000273616
I0922 17:05:04.050149 16604 solver.cpp:244] Train net output #0: loss = 0.000273616 (* 1 = 0.000273616
loss)
I0922 17:05:04.050149 16604 sgd_solver.cpp:106] Iteration 1100, lr = 0.00924715
I0922 17:05:13.845834 16604 solver.cpp:228] Iteration 1200, loss = 0.000260927
I0922 17:05:13.845834 16604 solver.cpp:244] Train net output #0: loss = 0.000260927 (* 1 = 0.000260927
loss)
I0922 17:05:13.846834 16604 sgd_solver.cpp:106] Iteration 1200, lr = 0.00918515
I0922 17:05:23.707509 16604 solver.cpp:228] Iteration 1300, loss = 0.000218446
I0922 17:05:23.707509 16604 solver.cpp:244] Train net output #0: loss = 0.000218446 (* 1 = 0.000218446
loss)
I0922 17:05:23.707509 16604 sgd_solver.cpp:106] Iteration 1300, lr = 0.00912412
I0922 17:05:33.580919 16604 solver.cpp:228] Iteration 1400, loss = 0.000253485
I0922 17:05:33.580919 16604 solver.cpp:244] Train net output #0: loss = 0.000253485 (* 1 = 0.000253485
loss)
I0922 17:05:33.580919 16604 sgd_solver.cpp:106] Iteration 1400, lr = 0.00906403
I0922 17:05:43.278966 16604 solver.cpp:337] Iteration 1500, Testing net (#0)
```

图 10.4 SiameseNet 手写数字验证案例程序训练的中间结果

```
C:\WINDOWS\system32\cmd.exe
I0921 19:04:36.430196 9064 solver.cpp:244] Train net output #0: loss = 1.51473e-006 (* 1 = 1.51473e-0
06 loss)
I0921 19:04:36.430196 9064 sgd_solver.cpp:106] Iteration 49600, lr = 0.00262159
I0921 19:04:46.144251 9064 solver.cpp:228] Iteration 49700, loss = 2.15816e-006
I0921 19:04:46.145251 9064 solver.cpp:244] Train net output #0: loss = 2.15816e-006 (* 1 = 2.15816e-0
06 loss)
I0921 19:04:46.145251 9064 sgd_solver.cpp:106] Iteration 49700, lr = 0.0026133
I0921 19:04:55.883679 9064 solver.cpp:228] Iteration 49800, loss = 9.86331e-007
I0921 19:04:55.883679 9064 solver.cpp:244] Train net output #0: loss = 9.86328e-007 (* 1 = 9.86328e-0
07 loss)
I0921 19:04:55.883679 9064 sgd_solver.cpp:106] Iteration 49800, lr = 0.00261501
I0921 19:05:05.570510 9064 solver.cpp:228] Iteration 49900, loss = 1.15899e-006
I0921 19:05:05.570510 9064 solver.cpp:244] Train net output #0: loss = 1.15899e-006 (* 1 = 1.15899e-0
06 loss)
I0921 19:05:05.570510 9064 sgd_solver.cpp:106] Iteration 49900, lr = 0.00261174
I0921 19:05:15.198707 9064 solver.cpp:454] Snapshotting to binary proto file F:\caffe-windows-master\caff
e-windows-master\examples\siamese\mnist_siamese_iter_50000.caffemodel
I0921 19:05:15.223702 9064 solver.cpp:273] Snapshotting solver state to binary proto file F:\caffe-wi
ndows-master\caffe-windows-master\examples\siamese\mnist_siamese_iter_50000_solverstate
I0921 19:05:15.291702 9064 solver.cpp:317] Iteration 50000, loss = 1.82972e-006
I0921 19:05:15.291702 9064 solver.cpp:337] Iteration 50000, Testing net (#0)
I0921 19:05:21.963867 9064 solver.cpp:404] Test net output #0: loss = 1.40023e-006 (* 1 = 1.40023e-00
6 loss)
I0921 19:05:21.964867 9064 solver.cpp:322] Optimization Done.
I0921 19:05:21.965867 9064 caffe.cpp:222] Optimization Done.

F:\caffe-windows-master\caffe-windows-master\bin>
```

图 10.5 SiameseNet 手写数字验证案例程序的最终训练和测试结果

10.2 挤压网络 SqueezeNet

10.2.1 SqueezeNet 的模型结构

卷积神经网络一般规模较大、参数较多，比如 AlexNet 总共有 8 层，有 65 万个神经元和 6000 万个参数。其他网络，像 VGGNet、ResNet 和 DenseNet 等，还可能更为复杂。简化模型的结构、压缩模型的参数，就成为改进卷积神经网络的必要问题。

为了利用较少的网络参数获得相近的性能，Iandola 等人提出了一种压缩结构，所产生的结果称为挤压网络 SqueezeNet。这种结构对卷积神经网络进行结构压缩，遵循下面 3 个

基本设计策略：

- 1) 使用 1×1 卷积核代替 3×3 卷积核；
- 2) 减少 3×3 卷积核的输入通道数量；
- 3) 推后下采样，增大卷积层的激活图 (activation map)。

在这 3 个策略中，前两个策略用于减少网络参数的总体预算，第 3 个策略用于在有限参数预算的条件下计算更多的卷积特征，综合起到保持甚至提高准确率的效果。基于这些策略，可以得到一种 AlexNet 的挤压网络模型，如图 10.6a 所示。其中输入结构为 $227 \times 227 \times 3$ ，表示一个大小为 227×227 的 3 通道彩色图像。另外，还包含两个独立卷积层 (conv1 和 conv10)、4 个池化层 (maxpool1 ~ 3 和 global avgpool)、8 个火焰模块 (fire2 ~ 9)，以及 1 个软最大输出层 (softmax)。

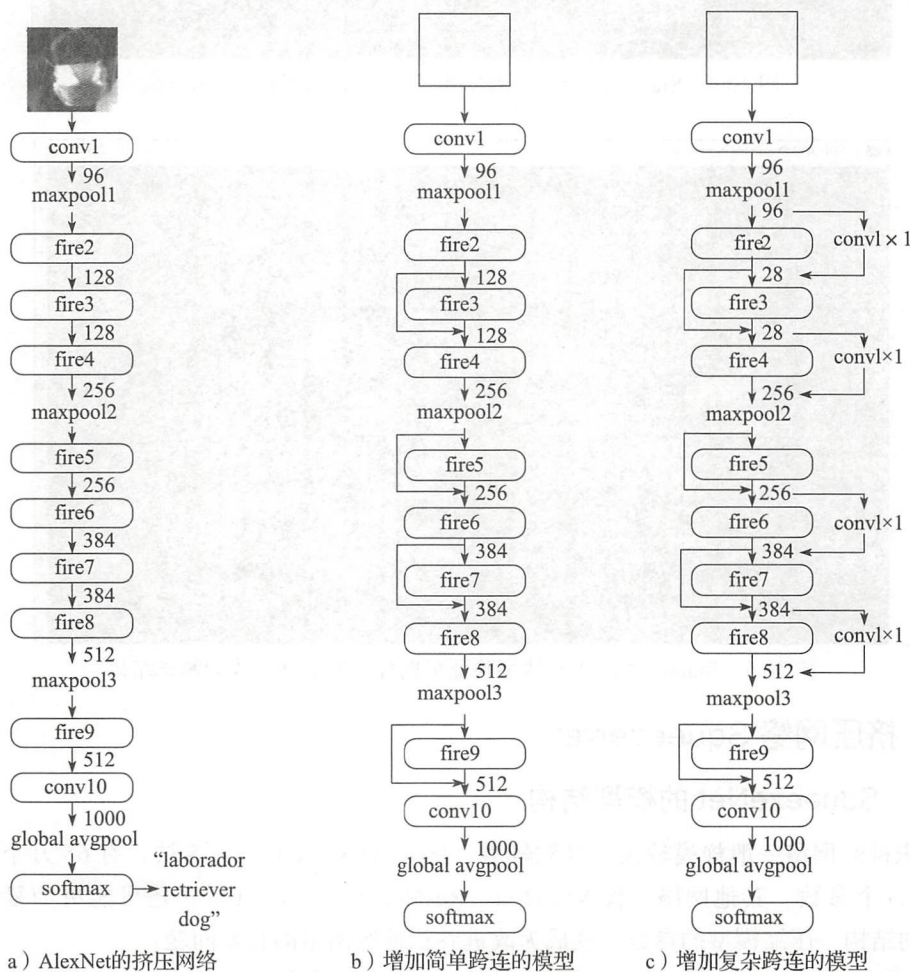


图 10.6 挤压网络的总体结构

SqueezeNet 的最大特色是火焰模块 (fire module)，其结构如图 10.7 所示。每个火焰模块实际上由两个卷积层组成，一个称为压缩层 (squeeze layer)，另一个称为扩展层 (expand layer)。压缩层只有 1×1 大小的卷积核 (记为 $S_{1 \times 1}$)，在经过 ReLU 函数变换后输入到扩展层。扩展层可以有 1×1 和 3×3 大小的两种卷积核 (分别记为 $e_{1 \times 1}$ 和 $e_{3 \times 3}$)，在经过 ReLU 函数后输入到下一层。在火焰模块中，一般要求 $S_{1 \times 1}$ 的数量小于 $e_{1 \times 1}$ 与 $e_{3 \times 3}$ 的数量之和，这样有助于限制卷积核的输入通道数量。表 10.1 ~ 表 10.3 给出了独立卷积层、池化层和火焰模块的细节描述。

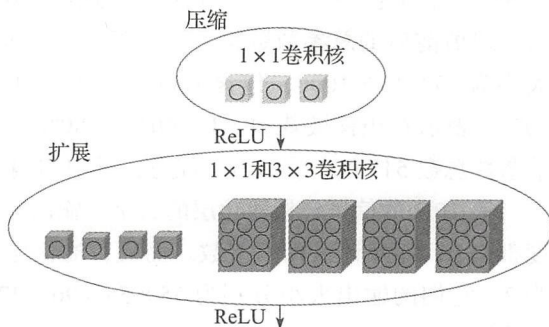


图 10.7 火焰模块示意图

表 10.1 独立卷积层的描述 (filter 可以指卷积核或池化窗口)

layer name	depth	output size	filter size/stride	sparsity	# pbp	# pap
conv1	1	$111 \times 111 \times 96$	$7 \times 7/2 (\times 96)$	100%	14 208	14 208
conv10	1	$13 \times 13 \times 1000$	$1 \times 1/1 (\times 1000)$	20%	513 000	103 400

表 10.2 池化层的描述 (filter 可以指卷积核或池化窗口)

layer name	depth	output size	filter size	stride
maxpool1	0	$55 \times 55 \times 96$	3×3	2
maxpool2	0	$27 \times 27 \times 256$	3×3	2
maxpool3	0	$13 \times 13 \times 512$	3×3	2
global avgpool	0	$1 \times 1 \times 1000$	13×13	1

表 10.3 火焰模块的描述

name	depth	output size	$S_{1 \times 1}$		$e_{1 \times 1}$		$e_{3 \times 3}$		# pbp	# pap
fire2	2	$55 \times 55 \times 128$	16	100%	64	100%	64	33%	11 920	5746
fire3	2	$55 \times 55 \times 128$	16	100%	64	100%	64	33%	12 432	6258
fire4	2	$55 \times 55 \times 256$	32	100%	128	100%	128	33%	45 344	20 646
fire5	2	$27 \times 27 \times 256$	32	100%	128	100%	128	33%	49 440	24 742
fire6	2	$27 \times 27 \times 384$	48	100%	192	50%	192	33%	104 880	44 700
fire7	2	$27 \times 27 \times 384$	48	50%	192	100%	192	33%	111 024	46 236
fire8	2	$27 \times 27 \times 512$	64	100%	256	50%	256	33%	188 992	77 581
fire9	2	$13 \times 13 \times 512$	64	50%	256	100%	256	30%	197 184	77 581

表 10.1 的信息包括独立卷积层的层名 (layer name)、深度 (depth)、输出大小 (output size)、卷积核大小 / 步长 (filter size/stride)、卷积核稀疏度 (sparsity)、裁剪前的参数数量 (# parameter before pruning, 即 #pbp) 和裁剪后的参数数量 (# parameter after pruning, 即

#pap)。从表 10.1 可以看出，独立卷积层 conv1 的深度为 1、输出大小为 $111 \times 111 \times 96$ ，包含 96 个大小为 7×7 、步长为 2 的卷积核。conv1 的稀疏度为 100%，表示卷积核没有被裁剪，裁剪前后的总参数数量不变，都是 14 208。独立卷积层 conv10 的深度也为 1，但输出大小为 $13 \times 13 \times 1000$ ，包含 1000 个大小为 1×1 、步长为 1 的卷积核。conv10 的稀疏度为 20%，表示卷积核被裁剪 $(1 - 20\%) = 80\%$ ，裁剪后的总参数数量变为 103 400，约为裁剪前参数总数 513 000 的 20%。注意，其中的偏置并没有被裁剪。

表 10.2 的信息包括池化层的名字、输出大小、池化窗口大小和步长。注意，池化层的深度都是 0，即不计入网络层数。而且，池化层 maxpool1 ~ 3 的窗口大小都为 3×3 ，步长都为 2。它们的输出大小分别为 $55 \times 55 \times 96$ 、 $27 \times 27 \times 256$ 和 $13 \times 13 \times 512$ 。最后，全局平均池化层 (global avgpool) 的窗口大小为 13×13 ，步长为 1，产生的输出大小为 $1 \times 1 \times 1000$ 。

表 10.3 的信息包括火焰模块的名字、深度、输出大小、压缩层的卷积核数量及稀疏度、扩展层的卷积核数量及稀疏度、裁剪前的参数数量 (#pbp) 和裁剪后的参数数量 (#pap)。例如，火焰模块 fire2 的深度为 2，输出大小为 $55 \times 55 \times 128$ ，卷积核数量及稀疏度分别为 16 和 100%， 1×1 卷积核数量及稀疏度分别为 64 和 100%， 3×3 卷积核数量及稀疏度分别为 64 和 33%，裁剪前的参数总数为 11 920，裁剪后的参数总数为 5746。

最后，借鉴残差网络的思想，挤压网络还可以通过增加跨层连接，形成跨连挤压网络，包括简单跨连挤压网络和复杂跨连挤压网络，分别如图 10.6b 和图 10.6c 所示。简单跨连挤压网络增加了跨越火焰模块 fire3、fire5、fire7 和 fire9 的跨层连接。复杂跨连挤压网络进一步增加了跨越火焰模块 fire2、fire4、fire6 和 fire8 的跨层连接。与无跨连的挤压网络相比，简单跨连挤压网络和复杂跨连挤压网络的性能均有所提升。

10.2.2 SqueezeNet 的 Caffe 代码实现及说明

挤压网络的 Caffe 代码的下载网址为 <https://github.com/DeepScale/SqueezeNet>，其中 SqueezeNet_v1.0 文件夹包含两个主要文件：train_val.prototxt 和 solver.prototxt。train_val.prototxt 用来定义挤压网络的结构，solver.prototxt 用来设置网络训练的超参数值。考虑到代码间的相似性，下面仅对独立卷积层 conv1、最大池化层 maxpool1、火焰模块 fire2 和全局平均池化层 global avgpool 进行详细介绍和说明。

1. 独立卷积层 conv1 的定义

```
layer {
  name: "conv1"           # 名字
  type: "Convolution"
  bottom: "data"          # 前一层
  top: "conv1"
  convolution_param {
    num_output: 96        # 卷积核的个数
    kernel_size: 7         # 卷积核的大小
```

```

        stride: 2                # 卷积核的移动步长
        weight_filler {
            type: "xavier"        # 权值初始化方式
        }
    }
}

```

2. 最大池化层 maxpool1 的定义

```

layer {
    name: "pool1"
    type: "Pooling"
    bottom: "conv1"                # 该层的前一层
    top: "pool1"
    pooling_param {
        pool: MAX                # 池化方式
        kernel_size: 3           # 池化窗口的大小
        stride: 2                # 池化窗口的步长
    }
}

```

3. 火焰模块 fire2 的定义

fire2 的压缩层

```

layer {
    name: "fire2/squeeze1x1"
    type: "Convolution"
    bottom: "pool1"
    top: "fire2/squeeze1x1"
    convolution_param {
        num_output: 16           # 压缩层的卷积核个数
        kernel_size: 1           # 压缩层的卷积核大小
        weight_filler {
            type: "xavier"
        }
    }
}

```

```

layer {
    name: "fire2/relu_squeeze1x1"
    type: "ReLU"                # 使用 ReLU 激活函数
    bottom: "fire2/squeeze1x1"
    top: "fire2/squeeze1x1"
}

```

fire2 的扩展层—— 1×1 卷积部分

```

layer {
    name: "fire2/expand1x1"
    type: "Convolution"
    bottom: "fire2/squeeze1x1"
    top: "fire2/expand1x1"
    convolution_param {
        num_output: 64           # 扩展层的  $1 \times 1$  卷积核个数
        kernel_size: 1
    }
}

```

```

        weight_filler {
            type: "xavier"
        }
    }
}
layer {
    name: "fire2/relu_expand1x1"
    type: "ReLU"
    bottom: "fire2/expand1x1"
    top: "fire2/expand1x1"
}
# fire2 的扩展层——3×3 卷积部分
layer {
    name: "fire2/expand3x3"
    type: "Convolution"
    bottom: "fire2/squeeze1x1"
    top: "fire2/expand3x3"
    convolution_param {
        num_output: 64 # 扩展层的 3×3 卷积核个数
        pad: 1 # 对输入的 4 边各扩展 1 个单位长度，并用 0 填充
        kernel_size: 3
        weight_filler {
            type: "xavier"
        }
    }
}
}
layer {
    name: "fire2/relu_expand3x3"
    type: "ReLU"
    bottom: "fire2/expand3x3"
    top: "fire2/expand3x3"
}

# fire2 的扩展层——拼接部分，将扩展层的 1×1 和 3×3 卷积的激活图进行拼接
layer {
    name: "fire2/concat"
    type: "Concat"
    bottom: "fire2/expand1x1"
    bottom: "fire2/expand3x3"
    top: "fire2/concat"
}

```

4. 全局平均池化层 global avgpool 的定义

```

layer {
    name: "pool10"
    type: "Pooling"
    bottom: "conv10"
    top: "pool10"
    pooling_param {
        pool: AVE # 池化方式
        global_pooling: true # 选择全局池化方式
    }
}

```


最后，还需要指出：

- 1) 上述代码只是整个挤压网络定义的一部分，仅对 4 个结构不同的层或模块进行了说明，其余部分的定义与之类似，详见 train_val.prototxt 文件。
- 2) 在实现火焰模块的扩展层时， 3×3 卷积部分需要先对输入的 4 边各扩展 1 个单位长度，并用 0 填充，再进行卷积运算，以确保与 1×1 卷积部分的输出具有相同的大小。
- 3) 在实现火焰模块的扩展层时，先分别使用 1×1 和 3×3 的卷积核对压缩层的特征图进行卷积，再将得到的特征图进行拼接。

10.2.3 SqueezeNet 大规模图像分类案例

本节描述一个利用 SqueezeNet 在 Caffe 框架下进行大规模图像分类的案例，其中用到的 ImageNet 数据集可以根据表 1.2 提供的地址下载。注意，这个数据集的图像是 .jpg 格式的，需要参考本书 3.6.1 节转换成 Caffe 支持的 LEVELDB 格式。此外，还需要按照方框 10.4 在 train_val.prototxt 文件中修改训练集和验证集的存放路径，并按照方框 10.5 在 solver.prototxt 文件中设置有关超参数，包括修改 train_val.prototxt 的路径。

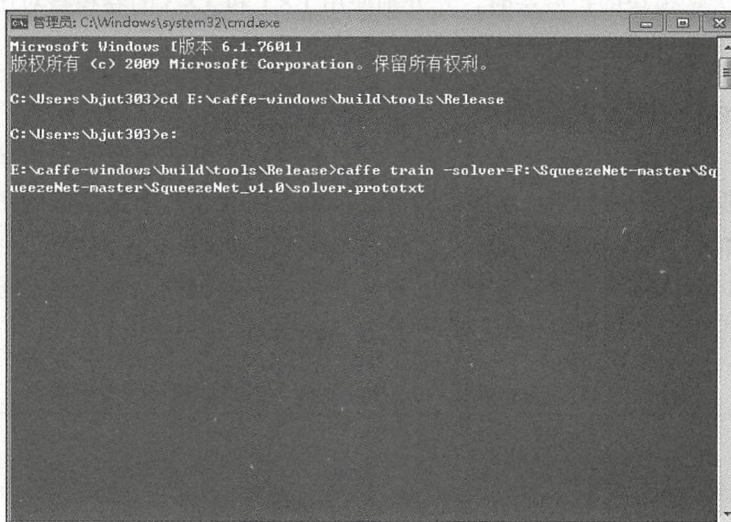
方框 10.4 在 train_val.prototxt 文件中的存放路径修改示例

```
Data_param{
  source:" F:/dataset/ilsvcr12/train_leveladb"
  batch_size:32
  backend:LEVELDB
}
Data_param{
  source:" F:/dataset/ilsvcr12/val_leveladb"
  batch_size:32
  backend:LEVELDB
}
```

方框 10.5 在 solver.prototxt 文件中的超参数设置情况

```
test_iter: 2000
test_interval: 1000
base_lr: 0.04
display: 40
max_iter: 170000
iter_size:16
lr_policy:" poly"
power: 1.0
momentum: 0.9
weight_decay: 0.0002
snapshot: 1000
snapshot_prefix: "train"
solver_mode: GPU
reandom_seed: 42
net: "F:/SqueezeNet-master/Squeeze-master/SqueezeNet_v1.0/train_val.prototxt"
test_initialization: false
average_loss: 40
```

在完成上述准备工作后，即可对 SqueezeNet 大规模图像分类案例程序进行训练。训练命令如图 10.8 所示，训练的中间结果如图 10.9 所示，最后结果如图 10.10 所示。如图 10.9 所示，在训练到 12 720 次时，学习率为 0.037 007 1，训练损失为 3.728 18。如图 10.10 所示，在训练结束时，此时共训练了 170 000 次，训练损失为 1.53 726，top-1 验证准确率约为 0.58，top-5 验证准确率约为 0.81。

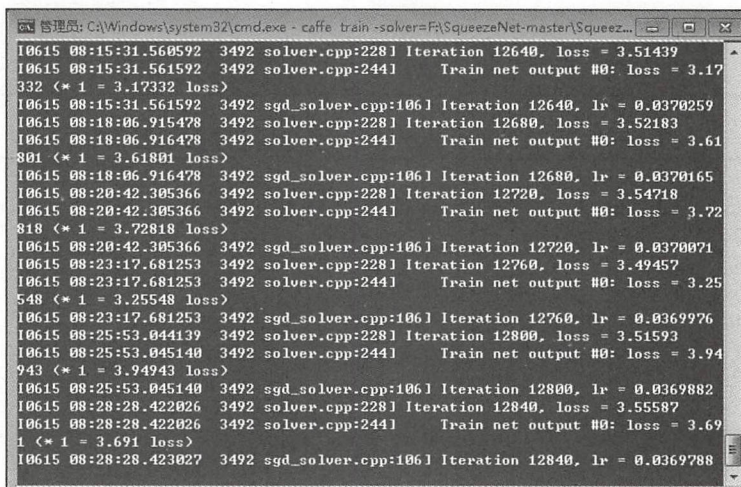


```

管理岗: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation. 保留所有权利。

C:\Users\hj303>cd E:\caffe-windows\build\tools\Release
C:\Users\hj303>e:
E:\caffe-windows\build\tools\Release>caffe train -solver=F:\SqueezeNet-master\SqueezeNet-master\SqueezeNet_v1.0\solver.prototxt
  
```

图 10.8 SqueezeNet 大规模图像分类案例程序的训练命令



```

管理岗: C:\Windows\system32\cmd.exe - caffe train -solver=F:\SqueezeNet-master\SqueezeNet-master\SqueezeNet_v1.0\solver.prototxt
10615 08:15:31.560592 3492 solver.cpp:228] Iteration 12640, loss = 3.51439
10615 08:15:31.561592 3492 solver.cpp:244] Train net output #0: loss = 3.17332 (* 1 = 3.17332 loss)
10615 08:15:31.561592 3492 sgd_solver.cpp:106] Iteration 12640, lr = 0.0370259
10615 08:18:06.915478 3492 solver.cpp:228] Iteration 12680, loss = 3.52183
10615 08:18:06.916478 3492 solver.cpp:244] Train net output #0: loss = 3.61801 (* 1 = 3.61801 loss)
10615 08:18:06.916478 3492 sgd_solver.cpp:106] Iteration 12680, lr = 0.0370165
10615 08:20:42.305366 3492 solver.cpp:228] Iteration 12720, loss = 3.54718
10615 08:20:42.305366 3492 solver.cpp:244] Train net output #0: loss = 3.72818 (* 1 = 3.72818 loss)
10615 08:20:42.305366 3492 sgd_solver.cpp:106] Iteration 12720, lr = 0.0370071
10615 08:23:17.681253 3492 solver.cpp:228] Iteration 12760, loss = 3.49457
10615 08:23:17.681253 3492 solver.cpp:244] Train net output #0: loss = 3.25548 (* 1 = 3.25548 loss)
10615 08:23:17.681253 3492 sgd_solver.cpp:106] Iteration 12760, lr = 0.0369976
10615 08:25:53.044139 3492 solver.cpp:228] Iteration 12800, loss = 3.51593
10615 08:25:53.045140 3492 solver.cpp:244] Train net output #0: loss = 3.94943 (* 1 = 3.94943 loss)
10615 08:25:53.045140 3492 sgd_solver.cpp:106] Iteration 12800, lr = 0.0369882
10615 08:28:28.422026 3492 solver.cpp:228] Iteration 12840, loss = 3.55587
10615 08:28:28.422026 3492 solver.cpp:244] Train net output #0: loss = 3.691 (* 1 = 3.691 loss)
10615 08:28:28.423027 3492 sgd_solver.cpp:106] Iteration 12840, lr = 0.0369788
  
```

图 10.9 SqueezeNet 大规模图像分类案例程序在训练 12 640 ~ 12 800 次时的中间结果


```

C:\Windows\system32\cmd.exe
10622 17:18:21.916091 3492 solver.cpp:2281 Iteration 169920, loss = 1.50543
10622 17:18:21.917091 3492 solver.cpp:2441 Train net output #0: loss = 1.53739 (* 1 = 1.53739 loss)
10622 17:18:21.917091 3492 sgd_solver.cpp:1061 Iteration 169920, lr = 1.88231e-05
10622 17:20:57.537992 3492 solver.cpp:2281 Iteration 169960, loss = 1.49993
10622 17:20:57.537992 3492 solver.cpp:2441 Train net output #0: loss = 1.34791 (* 1 = 1.34791 loss)
10622 17:20:57.537992 3492 sgd_solver.cpp:1061 Iteration 169960, lr = 9.41277e-06
10622 17:23:29.251670 3492 solver.cpp:4541 Snapshotting to binary proto file train_iter_170000.caffemodel
10622 17:23:29.701695 3492 sgd_solver.cpp:2731 Snapshotting solver state to binary proto file train_iter_170000.solverstate
10622 17:23:29.794701 3492 solver.cpp:3171 Iteration 170000, loss = 1.53726
10622 17:23:29.794701 3492 solver.cpp:3371 Iteration 170000, Testing net (#0)
10622 17:23:29.794701 3492 net.cpp:6931 Ignoring source layer loss
10622 17:25:32.023692 3492 solver.cpp:4041 Test net output #0: accuracy = 0.580259
10622 17:25:32.024693 3492 solver.cpp:4041 Test net output #1: accuracy_top5 = 0.806543
10622 17:25:32.024693 3492 solver.cpp:3221 Optimization Done.
10622 17:25:32.024693 3492 caffe.cpp:2551 Optimization Done.

E:\caffe-windows\build\tools\Release>

```

图 10.10 SqueezeNet 大规模图像分类案例程序在训练结束时的最终结果

10.3 深层卷积生成对抗网络 DCGAN

10.3.1 DCGAN 的模型结构

本节讨论的生成对抗网络 (Generative Adversarial Network, GAN) 是指深层卷积生成对抗网络 (Deep Convolutional GAN, DCGAN) [159]。理论上, 一个生成对抗网络包含两个均由多层神经网络构成的模型 [160]: 生成模型 (generative model) 和判别模型 (discriminative model)。生成模型, 或称生成器, 用 G 表示, 能够产生仿真数据分布。判别模型, 或称判别器, 用 D 表示, 用来判别数据是仿真的还是真实的。生成对抗网络的训练过程是使 G 产生的仿真数据尽可能逼近真实数据, 同时又使 D 尽量好地区分仿真数据和真实数据, 目标是使 G 产生的数据足以以假乱真, 使 D 判别真假的概率均为 0.5。

DCGAN 的基本思想是利用卷积神经网络替换普通的多层神经网络, 主要目的是生成尽量逼近真实图像的仿真图像。DCGAN 具有以下特点: ①判别器用卷积层代替池化层, 生成器用反卷积产生仿真图像; ②除了生成器的输出层和判别器的输入层外, 网络其他层都使用块归一化; ③生成器和判别器均无全连接层; ④生成器的输出层使用 Tanh 激活函数, 其他层使用 ReLU 激活函数; ⑤判别器使用 Leaky ReLU 激活函数。图 10.11 给出了一个 DCGAN 的生成器结构, 判别器使用与生成器相反的结构。生成器的输入是 100 维的向量, 但判别器只有两个输出神经元。表 10.4 详细描述了 DCGAN 的生成器结构, 包括层名、输出大小、卷积核大小 / 步长等。



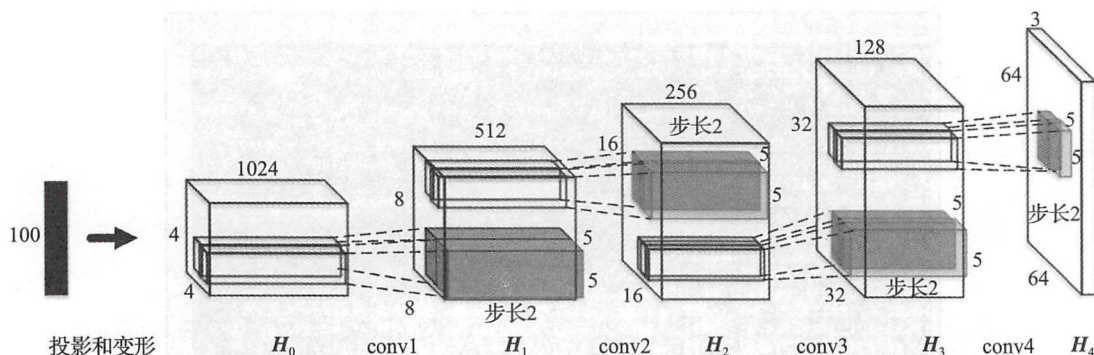


图 10.11 一个 DCGAN 的生成器结构

表 10.4 生成器结构的详细描述

layer name	type	output size	kernel size/stride
输入		100	
H_0	denselayer	$4 \times 4 \times 513$	
H_1	deconv	$8 \times 8 \times 256$	$5 \times 5/2$
H_2	deconv	$16 \times 16 \times 128$	$5 \times 5/2$
H_3	deconv	$32 \times 32 \times 64$	$5 \times 5/2$
H_4	deconv	$64 \times 64 \times 3$	$5 \times 5/2$

10.3.2 DCGAN 的 TensorFlow 代码实现及说明

DCGAN 的 TensorFlow 代码的下载地址为 <https://github.com/zsdonghao/dcgan>。在这个网址的文件夹 dcgan-master 下，包含求解器配置文件 main.py 和模型定义文件 model.py，下面分别详细介绍和说明。

1. 求解器配置文件 main.py

```
import os, sys, pprint, time
import scipy.misc
import numpy as np
import tensorflow as tf
import tensorlayer as tl
from tensorlayer.layers import *
from glob import glob
from random import shuffle
from model import *
from utils import *

pp = pprint.PrettyPrinter()
"""
TensorLayer implementation of DCGAN to generate face image.
```



Usage : see README.md

```

"""
flags = tf.app.flags
flags.DEFINE_integer("epoch", 25, "Epoch to train [25]")
flags.DEFINE_float("learning_rate", 0.0002, "Learning rate of for adam [0.0002]")
flags.DEFINE_float("beta1", 0.5, "Momentum term of adam [0.5]")
flags.DEFINE_integer("train_size", np.inf, "The size of train images [np.inf]")
flags.DEFINE_integer("batch_size", 64, "The number of batch images [64]")
flags.DEFINE_integer("image_size", 108, "The size of image to use (will be center
cropped) [108]")
flags.DEFINE_integer("output_size", 64, "The size of the output images to produce
[64]")
flags.DEFINE_integer("sample_size", 64, "The number of sample images [64]")
flags.DEFINE_integer("c_dim", 3, "Dimension of image color. [3]")
flags.DEFINE_integer("sample_step", 500, "The interval of generating sample. [500]")
flags.DEFINE_integer("save_step", 500, "The interval of saving checkpoints. [500]")
flags.DEFINE_string("dataset", "celebA", "The name of dataset [celebA, mnist, lsun]")
flags.DEFINE_string("checkpoint_dir", "checkpoint", "Directory name to save the
checkpoints [checkpoint]")
flags.DEFINE_string("sample_dir", "samples", "Directory name to save the image
samples [samples]")
flags.DEFINE_boolean("is_train", False, "True for training, False for testing [False]")
flags.DEFINE_boolean("is_crop", True, "True for training, False for testing [False]")
flags.DEFINE_boolean("visualize", True, "True for visualizing, False for
nothing [False]")
FLAGS = flags.FLAGS

def main(_):
    pp.pprint(flags.FLAGS.__flags)
    tl.files.exists_or_mkdir(FLAGS.checkpoint_dir)
    tl.files.exists_or_mkdir(FLAGS.sample_dir)
    z_dim = 100 # 输入维度
    with tf.device("/cpu:0"): # 使用 CPU 运行
        ##===== 定义模型 =====##
        z = tf.placeholder(tf.float32, [FLAGS.batch_size, z_dim], name='z_noise')
        real_images = tf.placeholder(tf.float32, [FLAGS.batch_size, FLAGS.output_
size, FLAGS.output_size, FLAGS.c_dim], name='real_images')
        # 将 z 输入到生成器进行训练
        net_g, g_logits = generator_simplified_api(z, is_train=True, reuse=False)
        # 将生成的仿真图像输入到判别器
        net_d, d_logits = discriminator_simplified_api(net_g.outputs, is_train=True,
reuse=False)
        # 将真实图像输入到判别器
        net_d2, d2_logits = discriminator_simplified_api(real_images, is_train=
True, reuse=True)
        # 采样 z 输入到生成器中进行测试
        # 使块归一化表现不同
        net_g2, g2_logits = generator_simplified_api(z, is_train=True, reuse=True)

        ##===== 定义训练选项 =====##
        # 更新判别器和生成器的代价
        # 判别器: 真实图像被标注为 1

```



```

d_loss_real = tl.cost.sigmoid_cross_entropy(d2_logits, tf.ones_like(d2_
logits), name='dreal')
# 判别器：生成器产生的图像被标注为 0
d_loss_fake = tl.cost.sigmoid_cross_entropy(d_logits, tf.zeros_like(d_
logits), name='dfake')
d_loss = d_loss_real + d_loss_fake
# 生成器：试图让伪造图像看起来像真实的

g_vars = tl.layers.get_variables_with_name('generator', True, True)
d_vars = tl.layers.get_variables_with_name('discriminator', True, True)

net_g.print_params(False)
print("-----")
net_d.print_params(False)

# 更新判别器和生成器
d_optim = tf.train.AdamOptimizer(FLAGS.learning_rate, beta1=FLAGS.beta1)
\minimize(d_loss, var_list=d_vars)
g_optim = tf.train.AdamOptimizer(FLAGS.learning_rate, beta1=FLAGS.beta1)
\minimize(g_loss, var_list=g_vars)

sess = tf.InteractiveSession()
tl.layers.initialize_global_variables(sess)

model_dir = "%s_%s_%s" % (FLAGS.dataset, FLAGS.batch_size, FLAGS.output_size)
save_dir = os.path.join(FLAGS.checkpoint_dir, model_dir)
tl.files.exists_or_mkdir(FLAGS.sample_dir)
tl.files.exists_or_mkdir(save_dir)
# 导入最新的 checkpoint
net_g_name = os.path.join(save_dir, 'net_g.npz')
net_d_name = os.path.join(save_dir, 'net_d.npz')

data_files = glob(os.path.join("C:\\Users\\ZhangTing\\Downloads\\dcgan-\\
master\\dcgan-master\\data\\celebA\\*.jpg"))

sample_seed = np.random.normal(loc=0.0, scale=1.0, size=(FLAGS.sample_size,
z_dim)).astype(np.float32)
# sample_seed = np.random.uniform(low=-1, high=1, size=(FLAGS.sample_size, z_
dim)).astype(np.float32)

##===== 训练模型 =====##
iter_counter = 0
for epoch in range(FLAGS.epoch):
    # 打乱数据
    shuffle(data_files)

    # 基于打乱数据更新样本文件
    sample_files = data_files[0:FLAGS.sample_size]
    sample = [get_image(sample_file, FLAGS.image_size, is_crop=FLAGS.is_crop,
resize_w=FLAGS.output_size, is_grayscale = 0) for sample_file in \
sample_files]
    sample_images = np.array(sample).astype(np.float32)
    print("[*] Sample images updated!")

```




```

# 导入图像数据
batch_idx = min(len(data_files), FLAGS.train_size) // FLAGS.batch_size

for idx in xrange(0, batch_idx):
    batch_files = data_files[idx*FLAGS.batch_size:(idx+1)*\
        FLAGS.batch_size]
    # 获得真实图像
    batch = [get_image(batch_file, FLAGS.image_size, is_crop=FLAGS.\
        is_crop, resize_w=FLAGS.output_size, is_grayscale = 0) for \
        batch_file in batch_files]
    batch_images = np.array(batch).astype(np.float32)
    batch_z = np.random.normal(loc=0.0, scale=1.0, size=(FLAGS.\
        sample_size, z_dim)).astype(np.float32) # batch_z = np.random.\
        uniform(low=-1, high=1, size=(FLAGS.batch_size, z_dim)).\
        astype(np.float32)
    start_time = time.time()
    # 更新判别器
    errD, _ = sess.run([d_loss, d_optim], feed_dict={z: batch_z,\
        z, real_images: batch_images })
    # 更新生成器, 运行生成器两次以保证判别器损失 d_loss 不为 0
    for _ in range(2):
        errG, _ = sess.run([g_loss, g_optim], feed_dict={z: batch_z})
        print("Epoch: [%2d/%2d] [%4d/%4d] time: %4.4f, d_loss:\
            %.8f, g_loss: %.8f" \
            % (epoch, FLAGS.epoch, idx, batch_idx, time.\
                time() - start_time, errD, errG))

    iter_counter += 1
    if np.mod(iter_counter, FLAGS.sample_step) == 0:
        # generate and visualize generated images 生成和可视化图像
        img, errD, errG = sess.run([net_g2.outputs, d_loss, g_loss],
            feed_dict={z : sample_seed, real_images: sample_images})
        tl.visualize.save_images(img, [8, 8], '.\\{}\train_{:02d}\_'\
            {:04d}.png'.format(FLAGS.sample_dir, epoch, idx))
        print("[Sample] d_loss: %.8f, g_loss: %.8f" % (errD, errG))

    if np.mod(iter_counter, FLAGS.save_step) == 0:
        # save current network parameters
        print("[*] Saving checkpoints...")
        tl.files.save_npz(net_g.all_params, name=net_g_name, sess=sess)
        tl.files.save_npz(net_d.all_params, name=net_d_name, sess=sess)
        print("[*] Saving checkpoints SUCCESS!")

if __name__ == '__main__':
    tf.app.run()

```

2. 模型定义文件 model.py

```

import tensorflow as tf
import tensorlayer as tl
from tensorlayer.layers import *

```



```

flags = tf.app.flags
FLAGS = flags.FLAGS

def generator_simplified_api(inputs, is_train=True, reuse=False): # 生成器的定义
    image_size = 64
    s2, s4, s8, s16 = int(image_size/2), int(image_size/4), int(image_size/8),
    int(image_size/16)
    gf_dim = 64 # Dimension of gen filters in first conv layer. [64]
    c_dim = FLAGS.c_dim # 彩色图像的维度
    batch_size = FLAGS.batch_size # 迷你块大小为 64
    w_init = tf.random_normal_initializer(stddev=0.02)
    gamma_init = tf.random_normal_initializer(1., 0.02)
    with tf.variable_scope("generator", reuse=reuse):
        tl.layers.set_name_reuse(reuse)

        net_in = InputLayer(inputs, name='g/in')
        net_h0 = DenseLayer(net_in, n_units=gf_dim*s16*s16, W_init=w_init,
                            act=tf.identity, name='g/h0/lin')
        net_h0 = ReshapeLayer(net_h0, shape=[-1, s16, s16, gf_dim*8], name='g/\
h0/reshape')
        net_h0 = BatchNormLayer(net_h0, act=tf.nn.relu, is_train=is_train,
                                gamma_init=gamma_init, name='g/h0/batch_norm')

        net_h1 = DeConv2d(net_h0, gf_dim*4, (5, 5), out_size=(s8, s8), strides=(2, 2),
                           padding='SAME', batch_size=batch_size, act=None, W_init=\
w_init, name='g/h1/decon2d')
        net_h1 = BatchNormLayer(net_h1, act=tf.nn.relu, is_train=is_train,
                                gamma_init=gamma_init, name='g/h1/batch_norm')

        net_h2 = DeConv2d(net_h1, gf_dim*2, (5, 5), out_size=(s4, s4), strides=(2, 2),
                           padding='SAME', batch_size=batch_size, act=None, W_init=w_\
init, name='g/h2/decon2d')
        net_h2 = BatchNormLayer(net_h2, act=tf.nn.relu, is_train=is_train,
                                gamma_init=gamma_init, name='g/h2/batch_norm')

        net_h3 = DeConv2d(net_h2, gf_dim, (5, 5), out_size=(s2, s2), strides=(2, 2),
                           padding='SAME', batch_size=batch_size, act=None, W_init=\
w_init, name='g/h3/decon2d')
        net_h3 = BatchNormLayer(net_h3, act=tf.nn.relu, is_train=is_train,
                                gamma_init=gamma_init, name='g/h3/batch_norm')

        net_h4 = DeConv2d(net_h3, c_dim, (5, 5), out_size=(image_size, image_\
size), strides=(2, 2), padding='SAME', batch_size=batch_size, act=None,
W_init=w_init, name='g/h4/decon2d')
        logits = net_h4.outputs
        net_h4.outputs = tf.nn.tanh(net_h4.outputs)
        return net_h4, logits

def discriminator_simplified_api(inputs, is_train=True, reuse=False):
    # 判别器的定义
    df_dim = 64 # Dimension of discrim filters in first conv layer. [64]
    c_dim = FLAGS.c_dim # n_color 3

```



```

batch_size = FLAGS.batch_size # 64
w_init = tf.random_normal_initializer(stddev=0.02)
gamma_init = tf.random_normal_initializer(1., 0.02)
with tf.variable_scope("discriminator", reuse=reuse):
    tl.layers.set_name_reuse(reuse)

    net_in = InputLayer(inputs, name='d/in')
    net_h0 = Conv2d(net_in, df_dim, (5, 5), (2, 2), act=lambda x: tl.act.lrelu(x, 0.2), padding='SAME', W_init=w_init, name='d/h0/conv2d')

    net_h1 = Conv2d(net_h0, df_dim*2, (5, 5), (2, 2), act=None, padding='SAME', W_init=w_init, name='d/h1/conv2d')
    net_h1 = BatchNormLayer(net_h1, act=lambda x: tl.act.lrelu(x, 0.2), is_train=is_train, gamma_init=gamma_init, name='d/h1/\batch_norm')

    net_h2 = Conv2d(net_h1, df_dim*4, (5, 5), (2, 2), act=None, padding='SAME', W_init=w_init, name='d/h2/conv2d')
    net_h2 = BatchNormLayer(net_h2, act=lambda x: tl.act.lrelu(x, 0.2), is_train=is_train, gamma_init=gamma_init, name='d/h2/\batch_norm')

    net_h3 = Conv2d(net_h2, df_dim*8, (5, 5), (2, 2), act=None, padding='SAME', W_init=w_init, name='d/h3/conv2d')
    net_h3 = BatchNormLayer(net_h3, act=lambda x: tl.act.lrelu(x, 0.2), is_train=is_train, gamma_init=gamma_init, name='d/h3/\batch_norm')

    net_h4 = FlattenLayer(net_h3, name='d/h4/flatten')
    net_h4 = DenseLayer(net_h4, n_units=1, act=tf.identity, W_init = w_init, name='d/h4/lin_sigmoid')
    logits = net_h4.outputs
    net_h4.outputs = tf.nn.sigmoid(net_h4.outputs)
    return net_h4, logits

```

10.3.3 DCGAN 的 CelebA 人脸图像生成案例

本节描述一个利用 DCGAN 在 TensorFlow 框架下进行人脸图像生成的案例，其中用到的 CelebA 数据集可以根据表 1.2 提供的地址下载。该案例利用 10.3.2 节的 main.py 和 model.py 程序，参照图 10.12 所示的命令运行（共迭代 25 次，迷你块为 3165 个）。程序运行过程中，随着迭代次数的增加，判别器和生成器的训练损失会不断发生变化。如图 10.13 所示，在迭代次数为 2（从次数 0 开始）、迷你块编号为 669（编号从 0 开始）时，判别器和生成器的训练损失分别为 1.301 426 65 和 0.729 599 36，生成的相应人脸图像例子详见图 10.14。如图 10.15 所示，在迭代次数为 13、迷你块编号为 2354 时，判别器和生成器的训练损失分别为 0.055 127 84 和 1.225 911 62，生成的相应人脸图像例子详见图 10.16。如图 10.17 所示，在程序运行结束时，判别器和生成器的训练损失分别为 0.127 069 94 和 2.981 523 51，生成的相应人脸图像例子详见图 10.18。




```

C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation. 保留所有权利。

C:\Users\ZhangTing>cd C:\Users\ZhangTing\Downloads\dcgan-master\dcgan-master
C:\Users\ZhangTing\Downloads\dcgan-master\dcgan-master>python main.py

```

图 10.12 DCGAN 人脸图像生成案例程序的运行命令

```

C:\WINDOWS\system32\cmd.exe - python main.py
Epoch: [ 2/25] [ 655/3165] time: 12.9354, d_loss: 1.36042929, g_loss: 0.70939839
Epoch: [ 2/25] [ 656/3165] time: 12.9679, d_loss: 1.32932222, g_loss: 0.73126483
Epoch: [ 2/25] [ 657/3165] time: 13.0357, d_loss: 1.32011962, g_loss: 0.73339404
Epoch: [ 2/25] [ 658/3165] time: 12.9749, d_loss: 1.37278390, g_loss: 0.59120433
Epoch: [ 2/25] [ 659/3165] time: 12.9661, d_loss: 1.43740463, g_loss: 0.99503404
Epoch: [ 2/25] [ 660/3165] time: 12.8977, d_loss: 1.50722153, g_loss: 0.52601039
Epoch: [ 2/25] [ 661/3165] time: 13.0128, d_loss: 1.37491989, g_loss: 0.31952918
Epoch: [ 2/25] [ 662/3165] time: 13.1570, d_loss: 1.33429987, g_loss: 0.71871237
Epoch: [ 2/25] [ 663/3165] time: 13.1983, d_loss: 1.30350554, g_loss: 0.71745896
Epoch: [ 2/25] [ 664/3165] time: 13.6477, d_loss: 1.26541066, g_loss: 0.73774438
Epoch: [ 2/25] [ 665/3165] time: 13.2522, d_loss: 1.41338360, g_loss: 0.68381512
Epoch: [ 2/25] [ 666/3165] time: 13.1110, d_loss: 1.33063574, g_loss: 0.81749445
Epoch: [ 2/25] [ 667/3165] time: 13.1785, d_loss: 1.33156319, g_loss: 0.70107508
Epoch: [ 2/25] [ 668/3165] time: 14.1245, d_loss: 1.36951828, g_loss: 0.79810739
Epoch: [ 2/25] [ 669/3165] time: 13.4601, d_loss: 1.30142665, g_loss: 0.72959936
[Sample] d_loss: 1.27123613, g_loss: 0.73857749
[*] Saving checkpoints...
[*] checkpoint/celebA_64_64/net_g.npz saved
[*] checkpoint/celebA_64_64/net_d.npz saved
[*] Saving checkpoints SUCCESS!
Epoch: [ 2/25] [ 670/3165] time: 13.1061, d_loss: 1.31507015, g_loss: 0.81276530
Epoch: [ 2/25] [ 671/3165] time: 12.9745, d_loss: 1.34362341, g_loss: 0.81319845
Epoch: [ 2/25] [ 672/3165] time: 12.9213, d_loss: 1.34692955, g_loss: 0.64042306
Epoch: [ 2/25] [ 673/3165] time: 12.8246, d_loss: 1.38340712, g_loss: 0.84439739
Epoch: [ 2/25] [ 674/3165] time: 12.9103, d_loss: 1.31224132, g_loss: 0.76689106
Epoch: [ 2/25] [ 675/3165] time: 12.8533, d_loss: 1.41107273, g_loss: 0.54625636
Epoch: [ 2/25] [ 676/3165] time: 12.7183, d_loss: 1.31428170, g_loss: 1.12381837
Epoch: [ 2/25] [ 677/3165] time: 12.9942, d_loss: 1.41738260, g_loss: 0.45967317
Epoch: [ 2/25] [ 678/3165] time: 13.5703, d_loss: 1.53699589, g_loss: 1.29907918
Epoch: [ 2/25] [ 679/3165] time: 13.4455, d_loss: 1.56699586, g_loss: 0.42576093

```

图 10.13 DCGAN 案例程序在迭代次数为 2 时的判别器和生成器的损失

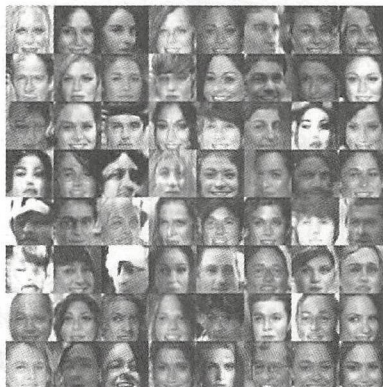


图 10.14 DCGAN 案例程序在运行迭代次数为 2 时生成的人脸图像

```

C:\WINDOWS\system32\cmd.exe - python main.py
Epoch: [13/25] [2340/3165] time: 13.2891, d_loss: 0.20533866, g_loss: 1.46852291
Epoch: [13/25] [2341/3165] time: 13.5591, d_loss: 0.06399764, g_loss: 1.10996461
Epoch: [13/25] [2342/3165] time: 13.8105, d_loss: 0.14052960, g_loss: 3.44338417
Epoch: [13/25] [2343/3165] time: 13.6777, d_loss: 0.08166433, g_loss: 2.87929296
Epoch: [13/25] [2344/3165] time: 15.9570, d_loss: 0.02655604, g_loss: 2.35577369
Epoch: [13/25] [2345/3165] time: 15.9530, d_loss: 0.17132637, g_loss: 0.66580552
Epoch: [13/25] [2346/3165] time: 14.7217, d_loss: 0.09003490, g_loss: 1.67934120
Epoch: [13/25] [2347/3165] time: 15.1046, d_loss: 0.40958129, g_loss: 3.44121695
Epoch: [13/25] [2348/3165] time: 13.5899, d_loss: 0.14470302, g_loss: 2.33947420
Epoch: [13/25] [2349/3165] time: 15.1380, d_loss: 0.11984592, g_loss: 0.60222965
Epoch: [13/25] [2350/3165] time: 14.5472, d_loss: 0.09907005, g_loss: 1.95207524
Epoch: [13/25] [2351/3165] time: 12.8752, d_loss: 0.10764112, g_loss: 3.96445107
Epoch: [13/25] [2352/3165] time: 13.6028, d_loss: 0.03253765, g_loss: 5.00523336
Epoch: [13/25] [2353/3165] time: 14.3227, d_loss: 0.19907324, g_loss: 1.20217597
Epoch: [13/25] [2354/3165] time: 13.6556, d_loss: 0.05512784, g_loss: 1.22591162
[Sample] d_loss: 0.19666289, g_loss: 2.47454023
[*] Saving checkpoints...
[*] checkpoint\celeba_64_64\net_g.npz saved
[*] checkpoint\celeba_64_64\net_d.npz saved
[*] Saving checkpoints SUCCESS!
Epoch: [13/25] [2355/3165] time: 13.9482, d_loss: 0.15027064, g_loss: 2.67366421
Epoch: [13/25] [2356/3165] time: 13.0818, d_loss: 0.03566094, g_loss: 2.83673024
Epoch: [13/25] [2357/3165] time: 12.8542, d_loss: 0.05609260, g_loss: 2.68073106
Epoch: [13/25] [2358/3165] time: 12.9304, d_loss: 0.10951979, g_loss: 1.07381968
Epoch: [13/25] [2359/3165] time: 12.8053, d_loss: 0.08496545, g_loss: 2.41386509
Epoch: [13/25] [2360/3165] time: 12.9278, d_loss: 0.05757813, g_loss: 2.09524870
Epoch: [13/25] [2361/3165] time: 12.9263, d_loss: 0.03440348, g_loss: 2.33134004
Epoch: [13/25] [2362/3165] time: 12.9813, d_loss: 0.05729807, g_loss: 0.91973501
Epoch: [13/25] [2363/3165] time: 12.8071, d_loss: 0.09189150, g_loss: 0.85838407
Epoch: [13/25] [2364/3165] time: 12.8036, d_loss: 0.11994246, g_loss: 1.62651038

```

图 10.15 DCGAN 案例程序在迭代次数为 13 时的判别器和生成器的损失

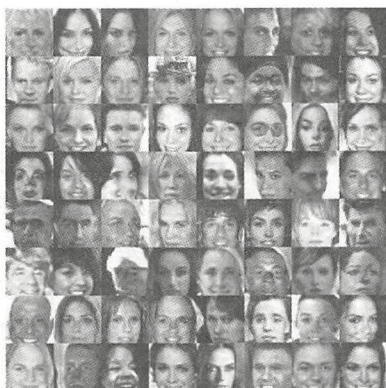


图 10.16 DCGAN 案例程序在迭代次数为 13 时生成的人脸图像

```

C:\WINDOWS\system32\cmd.exe
Epoch: [24/25] [3137/3165] time: 13.9609, d_loss: 0.05529121, g_loss: 3.79541302
Epoch: [24/25] [3138/3165] time: 14.0253, d_loss: 0.07751529, g_loss: 3.05340505
Epoch: [24/25] [3139/3165] time: 13.9420, d_loss: 0.05257650, g_loss: 3.53824139
Epoch: [24/25] [3140/3165] time: 13.8505, d_loss: 0.16372715, g_loss: 3.44475245
Epoch: [24/25] [3141/3165] time: 13.9499, d_loss: 0.05046088, g_loss: 3.61314982
Epoch: [24/25] [3142/3165] time: 13.8795, d_loss: 0.050384591, g_loss: 2.50814366
Epoch: [24/25] [3143/3165] time: 14.0162, d_loss: 0.13469732, g_loss: 3.51228380
Epoch: [24/25] [3144/3165] time: 13.8774, d_loss: 0.14128277, g_loss: 3.65430471
Epoch: [24/25] [3145/3165] time: 14.0214, d_loss: 0.85615987, g_loss: 0.27404159
Epoch: [24/25] [3146/3165] time: 13.9315, d_loss: 2.57266855, g_loss: 16.97504807
Epoch: [24/25] [3147/3165] time: 14.0330, d_loss: 2.73255749, g_loss: 3.32949471
Epoch: [24/25] [3148/3165] time: 13.9830, d_loss: 0.08092341, g_loss: 1.37817311
Epoch: [24/25] [3149/3165] time: 14.0117, d_loss: 0.61515182, g_loss: 7.57730007
Epoch: [24/25] [3150/3165] time: 13.9832, d_loss: 0.33972198, g_loss: 3.54590335
Epoch: [24/25] [3151/3165] time: 14.0224, d_loss: 0.09693579, g_loss: 2.33053614
Epoch: [24/25] [3152/3165] time: 13.9203, d_loss: 0.40042955, g_loss: 5.03401279
Epoch: [24/25] [3153/3165] time: 14.0707, d_loss: 0.29946190, g_loss: 2.39386368
Epoch: [24/25] [3154/3165] time: 13.9215, d_loss: 0.41321387, g_loss: 1.85712695
Epoch: [24/25] [3155/3165] time: 13.9637, d_loss: 0.20457278, g_loss: 3.47766876
Epoch: [24/25] [3156/3165] time: 14.0045, d_loss: 0.09494249, g_loss: 2.89902282
Epoch: [24/25] [3157/3165] time: 13.9297, d_loss: 0.07821666, g_loss: 3.44771671
Epoch: [24/25] [3158/3165] time: 13.7736, d_loss: 0.16426888, g_loss: 4.65731498
Epoch: [24/25] [3159/3165] time: 13.9279, d_loss: 0.44802174, g_loss: 1.01733442
Epoch: [24/25] [3160/3165] time: 14.0063, d_loss: 0.28933143, g_loss: 4.35278938
Epoch: [24/25] [3161/3165] time: 13.9378, d_loss: 0.09237669, g_loss: 4.78302141
Epoch: [24/25] [3162/3165] time: 13.8441, d_loss: 0.08136865, g_loss: 3.09452915
Epoch: [24/25] [3163/3165] time: 13.9192, d_loss: 0.13213639, g_loss: 1.87264001
Epoch: [24/25] [3164/3165] time: 13.9365, d_loss: 0.12706994, g_loss: 2.98152351
C:\Users\ZhangTing\Downloads\dcgan-master\dcgan-master>

```

图 10.17 DCGAN 案例程序在运行结束时判别器和生成器的损失



图 10.18 CelebA 人脸图像生成案例在运行结束后生成的人脸图像

10.4 网中网 NIN

10.4.1 NIN 的模型结构

网中网 (Network In Network, NIN) 是一种非常特殊的卷积网络模型, 其基本思想是使用一个微网络 (micro network) 卷积核来代替线性卷积核^[161]。微网络卷积核实际上是一个小型的多层感知器。与线性卷积核相比, 微网络卷积核在理论上能够近似任何局部的非线性函数变换, 不仅具有更高的抽象能力水平, 而且可以通过反向传播算法进行训练。

在网中网模型中, 可以使用两种卷积层: 一是图 10.19a 所示的线性卷积层, 二是图 10.19b 所示的多层感知器卷积层。在线性卷积层中, 每个卷积面都共享一个线性卷积核, 将输入局部块映射到输出特征图上的点。而在多层感知器卷积层中, 每个卷积面都共享一个微网络卷积核, 把输入局部块映射到输出特征图上的点。

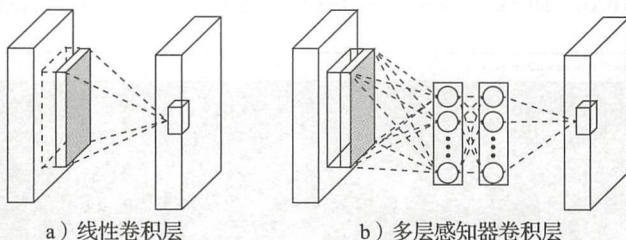


图 10.19 线性卷积层和多层感知器卷积层的比较。线性卷积层的每个卷积面共同使用线性卷积核计算特征图, 而多层感知器卷积层共同使用微网络卷积核计算特征图

从图 10.19 不难看出, 线性卷积层的结构相对简单, 所有局部感受野都通过一个线性卷积运算转变成特征图上的一个点。而多层感知器卷积层的结构相对复杂, 所有局部感受野都通过一个微网络卷积核转变成特征图上的一个点。如果用 x_{ij} 表示中心位置为 (i, j) 的局部感受野 (或局部块), $w_{k_l}^l$ 表示小型多层感知器第 l 层第 k_l 个神经元的权值, b_{k_l} 表示相应的偏置, f_{i,j,k_l}^l 表示相应的输出, 那么多层感知器卷积层计算特征图的过程可以表达如下:

$$\begin{cases} f_{i,j,k_1}^1 = \text{ReLU}((w_{k_1}^1)^T x_{i,j} + b_{k_1}) = \max((w_{k_1}^1)^T x_{i,j} + b_{k_1}, 0) \\ f_{i,j,k_2}^2 = \text{ReLU}((w_{k_2}^2)^T f_{i,j}^1 + b_{k_2}) = \max((w_{k_2}^2)^T f_{i,j}^1 + b_{k_2}, 0) \\ \vdots \\ f_{i,j,k_l}^l = \text{ReLU}((w_{k_l}^l)^T f_{i,j}^{l-1} + b_{k_l}) = \max((w_{k_l}^l)^T f_{i,j}^{l-1} + b_{k_l}, 0) \\ \vdots \\ f_{i,j,k_n}^n = \text{ReLU}((w_{k_n}^n)^T f_{i,j}^{n-1} + b_{k_n}) = \max((w_{k_n}^n)^T f_{i,j}^{n-1} + b_{k_n}, 0) \end{cases} \quad (10.2)$$

其中, n 表示小型多层感知器在不计输入时的层数。

此外, 网中网模型还用全局平均池化替代卷积神经网络的全连接层, 一方面大大减少了网络参数, 另一方面也有助于提高泛化能力。图 10.20 给出了一般网中网模型的结构示意图。图 10.21 给出了一个具体的网中网模型结构, 其详细描述见表 10.5。这个网中网模型共包含 12 个卷积层、4 个池化层和 1 个全连接层。其中, conv1、conv2、conv3、conv4 和 cccp1 ~ cccp8 是卷积层, pool1、pool2、pool3 和 pool4 是池化层, output 是全连接层。

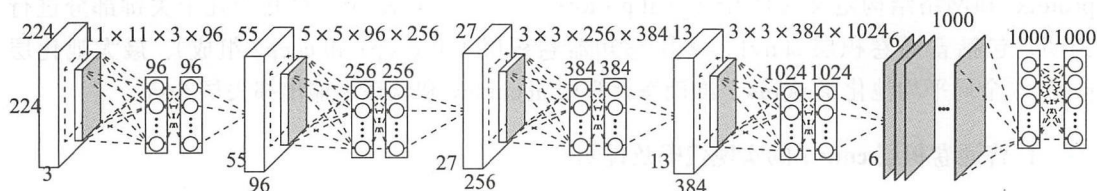


图 10.20 一般网中网模型的结构, 微网络卷积核的输入可能大于 1×1

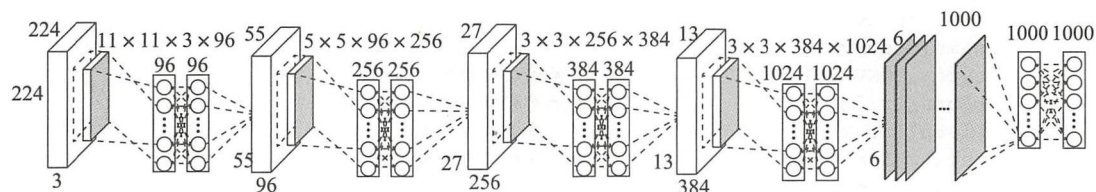


图 10.21 一个具体网中网模型的结构, 微网络卷积核的输入恰为 1×1

表 10.5 一个具体网中网模型结构的详细描述

layer name	output size	kernel size/number/pad/stride
数据	$224 \times 224 \times 3$	
conv1	$55 \times 55 \times 96$	$11 \times 11/96/3/4$
cccp1	$55 \times 55 \times 96$	$1 \times 1/96/0/1$
cccp2	$55 \times 55 \times 96$	$1 \times 1/96/0/1$
pool1	$27 \times 27 \times 96$	$3 \times 3///2$
conv2	$27 \times 27 \times 256$	$5 \times 5/256/2/1$
cccp3	$27 \times 27 \times 256$	$1 \times 1/256/0/1$
cccp4	$27 \times 27 \times 256$	$1 \times 1/256/0/1$
pool2	$13 \times 13 \times 256$	$3 \times 3///2$
conv3	$13 \times 13 \times 384$	$3 \times 3/384/1/1$

(续)

layer name	output size	kernel size/number/pad/stride
cccp5	$13 \times 13 \times 384$	$1 \times 1/384/0/1$
cccp6	$13 \times 13 \times 384$	$1 \times 1/384/0/1$
pool3	$6 \times 6 \times 384$	$3 \times 3//2$
conv4	$6 \times 6 \times 1024$	$3 \times 3/1024/1/1$
cccp7	$6 \times 6 \times 1024$	$1 \times 1/1024/0/1$
cccp8	$6 \times 6 \times 1024$	$1 \times 1/1000/0/1$
pool4	$1 \times 1 \times 1000$	$6 \times 6//1$
output	$1 \times 1 \times 1000$	

10.4.2 NIN 的 Caffe 代码实现及说明

网中网模型的 Caffe 实现代码的下载地址为 <https://gist.github.com/mavenlin/d802a5849de39225bcc6>。这个网址的子文件夹 Network in Network ILSVRC 包含求解器配置文件 solver.prototxt 和网络结构定义文件 train_val.prototxt。下面参考表 10.5 对其中几个关键部分进行说明，包括普通卷积层 conv1、多层感知器卷积层（由 cccp1 和 cccp2 组成）、最大池化层 pool1、全局平均池化层 pool4、准确率计算层 accuracy 和软最大输出损失层 loss。

1. 普通卷积层 conv1 的实现代码及说明

```

layers {
  bottom: "data"                # 底层为输入数据
  top: "conv1"
  name: "conv1"
  type: CONVOLUTION             # 表示该层为卷积层
  blobs_lr: 1                   # 表示卷积核的学习率
  blobs_lr: 2                   # 表示偏置的学习率
  weight_decay: 1               # 表示卷积核的衰减系数
  weight_decay: 0
  convolution_param {
    num_output: 96              # 输出特征图的个数
    kernel_size: 11             # 卷积核的大小
    stride: 4                   # 卷积核的步长
    weight_filler {
      type: "gaussian"          # 卷积核的初始化方式
      mean: 0
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}

layers {
  bottom: "conv1"

```

```

top: "conv1"
name: "relu0"
type: RELU                                # 把激活函数选为 ReLU
}

```

2. 多层感知器卷积层 cccp1 和 cccp2 的实现代码及说明

这个特殊层并不是通过专门的微网络卷积核来实现的，而是通过两个卷积核大小为 1×1 的卷积层 cccp1 和 cccp2 来实现的。需要强调的是，cccp1 的卷积核可以是 2×2 、 3×3 等不同大小，但 cccp2 的卷积核大小必须是 1×1 。否则，cccp1 和 cccp2 就不能实现一个等价的多层感知器卷积层。如果多层感知器卷积层的微网络卷积核层数增加，那么只需在 cccp2 之上再增加卷积核大小为 1×1 的卷积层即可。

```

layers {
  bottom: "conv1"                                # 底层为卷积层 conv1
  top: "cccp1"
  name: "cccp1"
  type: CONVOLUTION                                # 表示该层为卷积层
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  convolution_param {
    num_output: 96                                # 输出特征图的个数
    kernel_size: 1                                # 卷积核的大小
    stride: 1                                       # 卷积核的步长
    weight_filler {
      type: "gaussian"
      mean: 0
      std: 0.05
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}

layers {
  bottom: "cccp1"
  top: "cccp1"
  name: "relu1"
  type: RELU                                # 把激活函数选为 ReLU
}

layers {
  bottom: "cccp1"                                # 底层为卷积层 cccp1
  top: "cccp2"
  name: "cccp2"
  type: CONVOLUTION                                # 表示该层为卷积层
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
}

```



```

weight_decay: 0
convolution_param {
    num_output: 96          # 输出特征图的个数
    kernel_size: 1          # 卷积核的大小
    stride: 1              # 卷积核的步长
    weight_filler {
        type: "gaussian"
        mean: 0
        std: 0.05
    }
    bias_filler {
        type: "constant"
        value: 0
    }
}
}
layers {
    bottom: "cccp2"
    top: "cccp2"
    name: "relu2"
    type: RELU              # 把激活函数选为 ReLU
}

```

3. 最大池化层 pool1 的实现代码及说明

```

layers {
    bottom: "cccp2"
    top: "pool1"
    name: "pool1"
    type: POOLING
    pooling_param {
        pool: MAX          # 把池化方式选为最大池化
        kernel_size: 3     # 池化窗口的大小
        stride: 2          # 池化窗口的步长
    }
}

```

4. 全局平均池化层 pool4 的实现代码及说明

```

layers {
    bottom: "cccp8"
    top: "pool4"
    name: "pool4"
    type: POOLING
    pooling_param {
        pool: AVE          # 把池化方式选为平均池化
        kernel_size: 6     # 池化窗口的大小
        stride: 1          # 池化窗口的步长
    }
}

```

5. 准确率计算层 accuracy 的实现代码及说明

```

layers {

```

```

name: "accuracy"
type: ACCURACY          # 表示该层为准确率计算层
bottom: "pool4"         # 底层为池化层 pool4
bottom: "label"         # 底层为标签层 label
top: "accuracy"
include: { phase: TEST }
}

```

6. 软最大输出损失层 loss 的实现代码及说明

```

layers {
  bottom: "pool4"
  bottom: "label"
  name: "loss"
  type: SOFTMAX_LOSS    # 表示该层为软最大输出损失层
  include: { phase: TRAIN }
}

```

10.4.3 NIN 大规模图像分类案例

本节描述一个利用 NIN 在 Caffe 框架下进行大规模图像分类的案例，其中用到的 ImageNet 数据集可以根据表 1.2 提供的地址下载。注意，这个数据集的图像是 .jpg 格式的，需要参考本书 3.6.1 节转换成 Caffe 支持的 LEVELDB 格式。此外，还需要按照方框 10.6 在 train_val.prototxt 文件中修改训练集和验证集的存放路径，并按照方框 10.7 在求解器配置文件 solver.prototxt 中设置有关超参数，包括修改 train_val.prototxt 的路径。

在做好上述准备后，即可对 NIN 大规模图像分类案例程序进行训练和验证。运行命令如图 10.21 所示（共迭代 450 000 次），训练的中间结果如图 10.22 所示，最终结果如图 10.23 所示。如图 10.22 所示，在训练到 136 000 次时，学习率为 0.01，训练损失为 3.005 65，top-1 和 top-5 训练准确率分别为 37.5% 和 56.25%，top-1 和 top-5 验证准确率分别约为 35.09% 和 60.91%。如图 10.23 所示，在训练结束时，训练损失为 1.196 57，top-1 和 top-5 验证准确率分别约为 58.72% 和 81.53%。

方框 10.6 在 train_val.prototxt 文件中的存放路径修改示例

```

layers {
  top: "data"
  top: "label"
  name: "data"
  type: DATA
  data_param {
    source: "E:/dataset/ilsvcr12/train_leveladb"
    backend: LEVELDB
    batch_size: 64
  }
  transform_param {
    crop_size: 224
    mirror: true
    mean_file: "E:/dataset/ilsvcr12/mean.binaryproto"
  }
}

```

```

        include: { phase: TRAIN }
    }
    layers {
        top: "data"
        top: "label"
        name: "data"
        type: DATA
        data_param {
            source: " E:/dataset/ilsvc12/test_leveldb "
            backend: LEVELDB
            batch_size: 89
        }
    }
    transform_param {
        crop_size: 224
        mirror: false
        mean_file: " E:/dataset/ilsvc12/mean.binaryproto"
    }
    include: { phase: TEST }
}

```

方框 10.7 在 solver.prototxt 中设置的超参数情况

```

net: "F:/YongD/caffe-windows-master/models/nin/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 200000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: " F:/YongD/caffe-windows-master/models/nin/nin_imagenet_train"
solver_mode: GPU

```

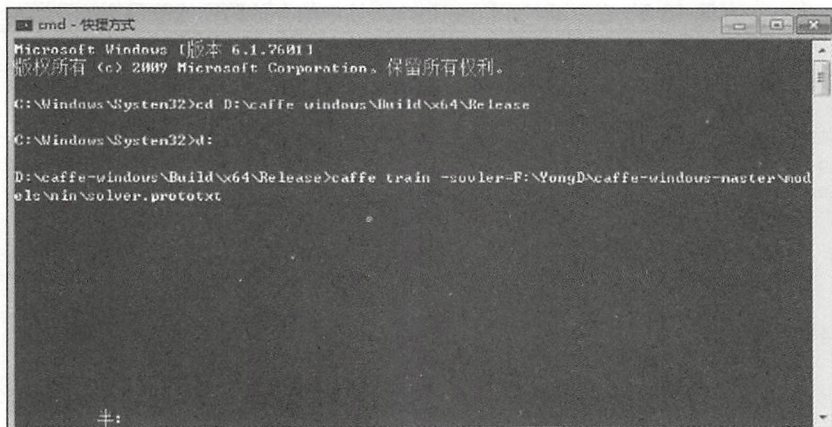


图 10.22 NIN 大规模图像分类案例程序的训练运行命令


```
cmd - 快捷方式 - caffe train -solver=F:\YongD\caffe-windows-master\models\nin\solver.prototxt
11212 17:11:39.276044 9124 sgd_solver.cpp:1061 Iteration 136000, lr = 0.01
11212 17:24:38.918614 9124 solver.cpp:3371 Iteration 136000, Testing net (#0)
11212 17:24:38.918614 9124 net.cpp:6841 Ignoring source layer loss
11212 17:28:24.526211 9124 solver.cpp:4041 Test net output #0: accuracy/top-1 = 0.3
50922
11212 17:28:24.526211 9124 solver.cpp:4041 Test net output #1: accuracy/top-5 = 0.6
09109
11212 17:28:24.760210 9124 solver.cpp:2281 Iteration 136000, loss = 3.00565
11212 17:28:24.760210 9124 solver.cpp:2441 Train net output #0: accuracy/top-1 = 0.
375
11212 17:28:24.760210 9124 solver.cpp:2441 Train net output #1: accuracy/top-5 = 0.
5625
11212 17:28:24.760210 9124 sgd_solver.cpp:1061 Iteration 136000, lr = 0.01
11212 17:41:29.269989 9124 solver.cpp:3371 Iteration 137000, Testing net (#0)
11212 17:41:29.269989 9124 net.cpp:6841 Ignoring source layer loss
11212 17:45:14.144383 9124 solver.cpp:4041 Test net output #0: accuracy/top-1 = 0.3
34312
11212 17:45:14.144383 9124 solver.cpp:4041 Test net output #1: accuracy/top-5 = 0.5
93875
11212 17:45:14.378384 9124 solver.cpp:2281 Iteration 137000, loss = 2.80061
11212 17:45:14.378384 9124 solver.cpp:2441 Train net output #0: accuracy/top-1 = 0.
359375
11212 17:45:14.378384 9124 solver.cpp:2441 Train net output #1: accuracy/top-5 = 0.
5625
11212 17:45:14.378384 9124 sgd_solver.cpp:1061 Iteration 137000, lr = 0.01
半:
```

图 10.23 NIN 大规模图像分类案例程序在训练 136 000 ~ 137 000 次时的中间结果

```
cmd - 快捷方式
11216 07:32:18.671052 9124 solver.cpp:4041 Test net output #0: accuracy/top-1 = 0.5
86125
11216 07:32:18.671052 9124 solver.cpp:4041 Test net output #1: accuracy/top-5 = 0.8
14203
11216 07:32:18.920653 9124 solver.cpp:2281 Iteration 449000, loss = 1.57902
11216 07:32:18.920653 9124 solver.cpp:2441 Train net output #0: accuracy/top-1 = 0.
640625
11216 07:32:18.920653 9124 solver.cpp:2441 Train net output #1: accuracy/top-5 = 0.
84375
11216 07:32:18.920653 9124 sgd_solver.cpp:1061 Iteration 449000, lr = 0.0001
11216 07:45:22.900030 9124 solver.cpp:4541 Snapshotting to binary proto file F:\YongD\c
affe-windows-master\models\nin\nin_imagenet_train_iter_450000.caffemodel
11216 07:45:23.586431 9124 sgd_solver.cpp:2731 Snapshotting solver state to binary prot
o file F:\YongD\caffe\windows-master\models\nin\nin_imagenet_train_iter_450000.solversta
te
11216 07:45:23.898432 9124 solver.cpp:3171 Iteration 450000, loss = 1.19657
11216 07:45:23.898432 9124 solver.cpp:3371 Iteration 450000, Testing net (#0)
11216 07:45:23.898432 9124 net.cpp:6841 Ignoring source layer loss
11216 07:49:09.662029 9124 solver.cpp:4041 Test net output #0: accuracy/top-1 = 0.5
82203
11216 07:49:09.662029 9124 solver.cpp:4041 Test net output #1: accuracy/top-5 = 0.8
15313
11216 07:49:09.662029 9124 solver.cpp:3221 Optimization Done.
11216 07:49:09.662029 9124 caffe.cpp:2231 Optimization Done.
D:\caffe-windows\Build\x64\Release>
半:
```

图 10.24 NIN 大规模图像分类案例程序在训练结束时的最终结果

卷积神经网络的强化模型

通过与强化学习结合,卷积神经网络可以帮助智能体根据感知数据优化自己的行为,以提高应对环境变化的能力。这种结合产生的模型称为深度 Q 网络,又称深度强化学习或深层强化学习网络。深度强化学习网络是一种重要的深度学习模型,也是人工智能领域的一个研究热点。基于深度强化学习网络的智能体可以通过端到端的学习方式在高维原始输入数据和其行为活动之间建立复杂的驱动关联,从而指导其完成高难度的控制决策任务,比如在计算机游戏中战胜人类专业玩家。本章将介绍深度强化学习网络的有关概念、模型和算法,并讨论如何实现一个著名游戏——笨笨鸟 (Flappy Bird) 的智能体案例。

11.1 强化学习的基本概念

强化学习有时也称为增强学习,通常与序列决策有关,可用于解决科学、工程和艺术等领域的大量实际问题。简单地说,强化学习是一种从环境状态映射到动作行为的学习过程,目标是使智能体在与环境的交互过程中获得最大的累积奖赏^[162]。强化学习不同于监督学习,主要表现在教师信号上的差别。强化学习所能利用的信号相对较弱,只是一种对动作行为的好坏评价(通常为标量信号),而不是如何去产生正确的动作。由于外部环境提供的信息很少,所以一个强化学习系统,或者强化学习智能体,必须靠自身的经历进行学习,不断在动作-评价的过程中改进行为以适应环境。

对强化学习问题建模一般需要利用马尔可夫决策过程 (Markov Decision Process, MDP)。理论上,马尔可夫决策过程可以定义为一个四元组 (S, A, ρ, p) , 其中:

- 1) S 是环境状态的集合,智能体在时刻 t 的环境状态表示为 $s_t \in S$;
- 2) A 是动作空间的集合,智能体在时刻 t 所采取的动作表示为 $a_t \in A$;
- 3) $\rho: S \times A \rightarrow R$ 是奖赏函数,智能体在状态 s_t 执行动作 a_t 获得的立即奖赏值 $r_t = \rho(s_t, a_t)$;
- 4) $p: S \times S \times A \rightarrow [0, 1]$ 是状态转移概率分布函数,智能体在状态 s_t 执行动作 a_t 后转移到下一状态 s_{t+1} 的概率表示为 $p(s_{t+1} | s_t, a_t)$ 。

另外, 未来的即时回报应该乘以一个折扣因子 $\gamma \in [0, 1]$, 以平衡即时回报对累积奖赏的影响。这是因为越往后, 越接近目标, 奖赏对于智能体的整体回报的贡献越小。 γ 的值越大, 说明每一次探索对智能体达到目标的重要性越大。 $\gamma = 0$ 表示智能体只考虑“眼前利益”, 根本不考虑长远利益, $\gamma = 1$ 表示智能体在未来的每一次探索对最终回报都具有同等重要性。因此, 从 t 刻到 T 时刻的累积奖赏可定义为

$$R_t = \sum_{t'=t}^T \gamma^{t'-t} r_{t'} \quad (11.1)$$

强化学习的基本原理是: 如果智能体的某个行为策略导致环境正的奖赏, 那么这个行为策略的使用趋势就会加强。图 11.1 是智能体进行强化学习的示意图。

强化学习的目标是建立一个从环境状态到动作空间的最优行为策略 $\pi: S \rightarrow A$ 。根据该策略, 智能体在状态 s_t 执行的动作表示为 $a_t = \pi(s_t)$, 所获得的累积奖赏回报称为状态动作值函数, 表示为

$$Q^\pi(s, a) = E[R_t | s_t = s, a_t = a, \pi] \quad (11.2)$$

当 $Q^\pi(s, a)$ 达到最大值 $Q^*(s, a)$ 时, 相应的策略 π^* 称为最优状态动作策略。这种最优策略可能不止一个, 但共享最优状态动作值函数, 或称 Q 函数:

$$Q^*(s, a) = \max_{\pi} E[R_t | s_t = s, a_t = a, \pi] \quad (11.3)$$

不难证明, 最优状态动作值函数遵循贝尔曼方程 (Bellman equation), 即

$$Q^*(s, a) = E_{s' \sim s} [r + \gamma \max_{a'} Q(s', a') | s, a] \quad (11.4)$$

在传统的强化学习中, $Q^\pi(s, a)$ 一般通过迭代贝尔曼最优性方程来计算:

$$Q_{i+1}(s, a) = E_{s' \sim s} [r + \gamma \max_{a'} Q_i(s', a') | s, a] \quad (11.5)$$

其中, $Q^*(s, a) = \lim_{i \rightarrow \infty} Q_i(s, a)$, 在收敛时得到的最优状态动作策略为

$$\pi^* = \arg \max_{a \in A} Q^*(s, a) \quad (11.6)$$

用上述方法求解实际问题的最优行为策略往往是不可行的, 主要原因是, 在状态空间较大时所需计算量太大。一种常用的解决办法是采用线性函数逼近器来近似表示最优状态动作值函数。此外, 也可以采用深层神经网络等非线性函数逼近器来近似表示状态动作值函数, 而由此得到的模型就称为强化学习网络。

2015 年, Mnih 等人提出了深度 Q 网络 (Deep Q -Network, DQN), 开启了深度强化学习的研究^[38]。深度 Q 网络, 又称为深度强化学习网络 (或者深层强化学习网络), 用于把原始观察数据映射到动作行为, 其标准模型在结构上无异于卷积神经网络, 如图 11.2 所示。这个卷积神经网络的输入大小是 $84 \times 84 \times 4$, 由 4 帧等间隔或连续游戏画面图像的原始观察数据经过预处理归一化得到。输入之后是 2 个卷积层和 2 个全连层。全连层之后是输出层, 用于估算动作行

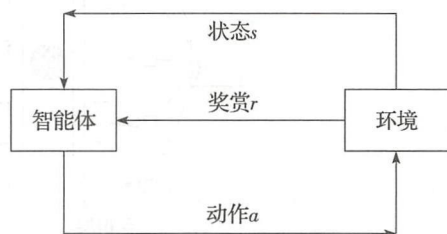


图 11.1 强化学习示意图

为的 Q 值，相应于控制游戏执行的各种操作（比如游戏杆的 17 种操作）。在深度强化学习网络中，环境状态被表达成一个原始观察数据 $\mathbf{x} \in R^d$ 和动作行为的序列，即 $s_t = \mathbf{x}_1, a_1, \mathbf{x}_2, \dots, a_{t-1}, \mathbf{x}_t$ 。深度强化学习网络与卷积神经网络的主要区别是，在学习训练的过程中，前者只给奖赏信号，并不指定具体的动作行为，而后者需要对一组给定的原始观察数据指明相应的动作行为。

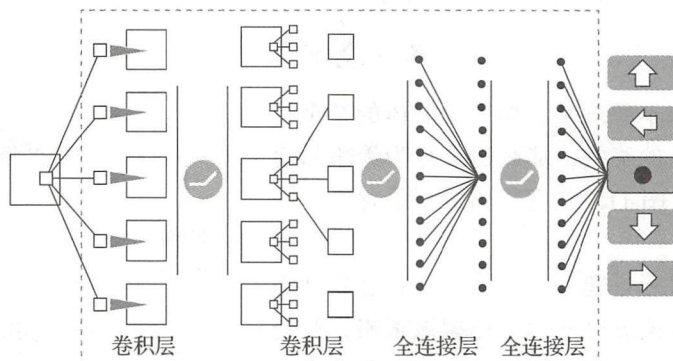


图 11.2 深度强化学习的卷积神经网络结构

深度强化学习网络在本质上是一种利用卷积神经网络来近似表示最优状态动作值函数以进行强化学习的模型。这个卷积神经网络可以表示为 $Q(s, a; \theta) \approx Q^*(s, a)$ ，其中 θ 表示网络的所有可调参数。如果用 θ^- 表示以前的网络参数，并定义目标 Q 值变量：

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-) \quad (11.7)$$

那么深度强化学习网络所采用的目标函数可以表示为

$$L(\theta) = E_{s,a,r} [(E_{s'}[y|s,a] - Q(s,a;\theta))^2] = E_{s,a,r,s'} [(y - Q(s,a;\theta))^2] - E_{s,a,r} [V_{s'}[y]] \quad (11.8)$$

其中， E 表示求期望， V 表示求方差。

显然，对式 (11.7) 的优化等价于优化下面的目标函数：

$$L(\theta) = E_{s,a,r,s'} [(y - Q(s,a;\theta))^2] \quad (11.9)$$

11.2 深度强化学习网络的学习算法

传统强化学习在利用非线性函数（比如神经网络）近似 Q 函数时，训练过程常常是不稳定的，甚至是发散的。深度 Q 网络提出的动机之一就是解决训练不稳定的问题。关键是如何有效地估算目标函数 $L(\theta)$ 的梯度。对式 (11.9) 的参数 θ 求偏导，得到梯度：

$$\nabla_{\theta} L(\theta) = E_{s,a,r,s'} [(y - Q(s,a;\theta)) \nabla_{\theta} Q(s,a;\theta)] \quad (11.10)$$

深度 Q 网络的学习算法伪码详见算法 11.1，其主要贡献在于如下 3 点：

1) 使用经验回放 (experience replay) 稳定 Q 函数训练过程。用 $e_t = (s_t, a_t, r_t, s_{t+1})$ 表示智能体在时刻 t 获得的经验，用 $D_t = \{e_1, e_2, \dots, e_t\}$ 存储这些经验。在训练时，每次从 D_t 中

随机抽取小批量的经验样本，先根据式 (11.10) 估算梯度 $\nabla_{\theta} L(\theta)$ ，再更新网络参数 θ 。

2) 设计了一种端到端的随机梯度下降算法。这种算法只需输入图像像素和游戏得分，对领域知识的需求极少，其中根据 ϵ 贪婪策略选择行为。

3) 具有很好的灵活性和自适应性。采用同样的算法、结构和超参数，在许多不同的任务上，特别是在 49 个 Atari 游戏上^[163]，都获得了成功训练，效果上超过了以前的算法，并达到了与人类专业测试者相媲美的竞技水平。

算法 11.1 DQN 学习算法

输入：图像像素、游戏得分

输出：网络参数 θ^- 、用于选择行为策略的 Q 函数

1. 初始化回放内存 D ，随机初始化 θ ，令 $\theta^- = \theta$;
2. **for** episode = 1 to M **do**
3. 初始化序列 $s_1 = \{x_1\}$ 和前处理序列 $\phi_1 = \phi(s_1)$;
4. **for** $t = 1$ to T **do**
5. 根据 ϵ 贪婪策略，以概率 ϵ 随机选择 a_t ，否则 $a_t = \arg \max_a Q(\phi(s_t), a; \theta)$;
6. 在仿真器中执行 a_t ，观察奖赏 r_t 和图像 x_{t+1} ;
7. 令 $s_{t+1} = s_t$ ， a_t ， x_{t+1} ，前处理序列 $\phi_{t+1} = \phi(s_{t+1})$;
8. 将转移 $(\phi_t, a_t, s_t, \phi_{t+1})$ 存入 D ;
9. 从 D 中随机采样一小批转移 $(\phi_j, a_j, s_j, \phi_{j+1})$;
10. 如果在 $j+1$ 处结束，令 $y_j = r_j$ ，否则令 $y_j = r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$;
11. 对目标函数 $(y_j - Q(\phi_j, a_j; \theta))^2$ 关于网络参数 θ 执行梯度下降过程;
12. 每执行 C 次梯度下降，更新 $\theta^- = \theta$;
13. **end for**
14. **end for**

11.3 深度强化学习网络的变种模型

深度强化学习网络的主要变种包括双重深度 Q 网络 (Double DQN, DDQN)^[164]、深度循环 Q 网络 (Deep Recurrent Q-Network, DRQN)^[165]、竞争 Q 网络 (dueling Q-network)^[166]、层次化深度 Q 网络 (hierarchical DQN, h-DQN)^[167]、记忆深度 Q 网络 (Memory Q-Network, MQN)^[168] 等。

DDQN 的提出是为了解决 DQN 对动作行为 Q 值的过高估计问题。与 DQN 相比，DDQN 的主要区别在于定义了一个具有两套不同参数的新目标 Q 值变量：

$$y = r + \gamma Q(s', \arg \max_a Q(s', a; \theta); \theta^-) \quad (11.11)$$

其中，第一套参数用来选择对应最大 Q 值的动作，第二套参数用来评估最优动作的 Q 值。基于这种新的目标 Q 值变量，DDQN 能够估计出更加准确的 Q 值，并且可在一些 Atari 2600 游戏中获得更稳定有效的状态动作策略。

DRQN 是为了克服 DQN 的有限记忆能力而提出的模型。这种模型可以在 DQN 结构中把第一个全连接层替换成循环长短期记忆模块而得到，如图 11.3 所示。在本质上，DRQN 是一种卷积神经网络、循环神经网络和强化学习的有机结合。与 DQN 相比，DRQN 的优点是，能够在每个时刻把当前输入的图像保留一段时间，从而可以利用较少的计算资源通过时间循环把多帧图像的信息进行集成。如果在部分状态可观察的情况下训练、用越来越完全可观察的数据来评测，DRQN 的性能优于 DQN 网络。反之，如果在完全可观察的情况下训练、用部分可观察的数据来评测，DRQN 的性能则劣于 DQN 网络。

竞争 Q 网络与深度 Q 网络的不同之处在于将卷积神经网络提取的抽象特征分流到两个支路，其中一路用来估计状态值函数，另一路用来估计动作优势值函数。这两路在前面共享卷积特征学习模块，在后面又通过一个汇聚层结合产生对状态动作值函数的估计，如图 11.4 所示。与深度 Q 网络相比，基于竞争 Q 网络的智能体在多个动作具有相同状态值的情形下可以获得明显的性能提升。

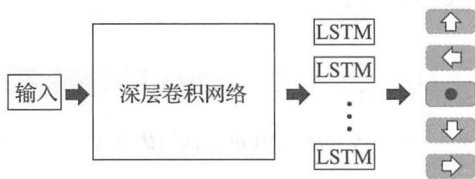
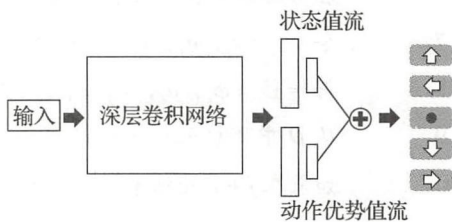


图 11.3 DRQN 的模型结构

图 11.4 竞争 Q 网络的模型结构

h-DQN 可以用来解决反馈极少的复杂目标导向型任务，比如经典的 Atari 游戏“Montezuma’s Revenge”。在这种任务中，智能体得到的奖赏反馈不仅是稀疏的而且是滞后的，所以很难对环境状态空间进行充分的探索和有效的学习。一种解决方案就是采用 h-DQN 在不时间尺度上对复杂任务进行分解，并通过时空抽象和内在激励设置子目标来层次化值函数。h-DQN 包含一个两阶段的分层结构，由控制器和元控制器组成，如图 11.5 所示。其中，元控制器用于确定智能体的决策，并选择下一个内在激励的子目标（比如寻找某个物体或对象），而控制器用于确定智能体的动作，以达到这一子目标。控制器和元控制器都可以用卷积神经网络来实现。此外，为了判断子目标是否达到，还需要定义一个评论员（比如关于物体布局关系的函数）以便在子目标达到时产生相应的

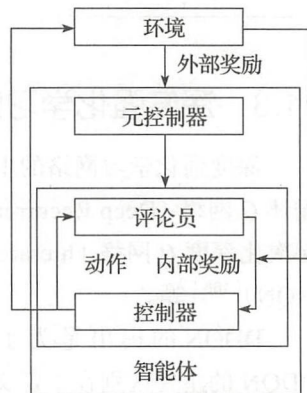


图 11.5 h-DQN 的模型结构

内部奖赏。

记忆深度 Q 网络的动机是提高智能体的认知、学习和推理能力。根据是否加入循环神经网络模块和反馈控制机制，可以分为记忆深度 Q 网络、循环记忆深度 Q 网络（Recurrent Memory Q -Network, RMQN）、反馈循环记忆深度 Q 网络（Feedback Recurrent Memory Q -Network, FRMQN）。这些模型与 DQN 和 DRQN 的结构对比如图 11.6 所示，其中上下文相当于是全连接层通过卷积神经网络从多幅图像中提取的特征或者通过全连接层特征的记忆、循环和反馈等机制提取的特征。与 DQN 和 DRQN 相比，MQN、RMQN 和 FRMQN 在一些未经训练的 Minecraft 游戏任务中具有更强的泛化能力。

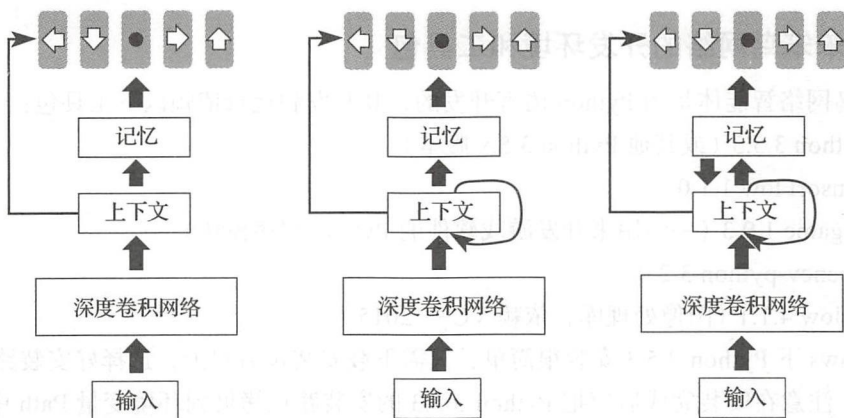


图 11.6 三种记忆强化网络对比图

11.4 深度强化学习网络的 Flappy Bird 智能体案例

本案例旨在讨论如何利用深度强化学习网络设计一个游戏智能体，自动控制 Flappy Bird 的飞行。

Flappy Bird（可译为“笨笨鸟”“飞飞鸟”“笨鸟先飞”等）是一款由越南游戏工程师 Dong Nguyen 开发的游戏作品，曾在苹果以及谷歌应用商店上线，下载量超过 5000 万次。现在既有手机版也有 PC 版，互联网上有很多介绍，感兴趣的读者可从网上下载。

如图 11.7a 所示，这款游戏的界面非常简洁，没有炫酷的效果，也没有过多的关卡。游戏玩家只需要通过手指点击屏幕来控制小鸟的动作。点击屏幕，小鸟就会以一定的速度往上飞翔，反之则会往下落。如果小鸟持续飞行，在不撞到地面并且穿过前方管道空隙，小鸟得分，如图 11.7b 所示。小鸟只有一条“命”，一旦撞到地面或者管道，游戏结束，如图 11.7c ~ e 所示。

人类玩家在玩这款游戏时，是很有难度和挑战性的，一般只能得 10 分左右，30 分就算高了，100 分就是很高的分数了。而如果利用深度强化学习网络来设计一个智能体，则所得分数可以远远超过人类玩家，轻松得到 1000 分以上。为了方便描述，本书将这个智能体称为笨笨鸟网络（Flappy Bird DQN, FBDQN）。

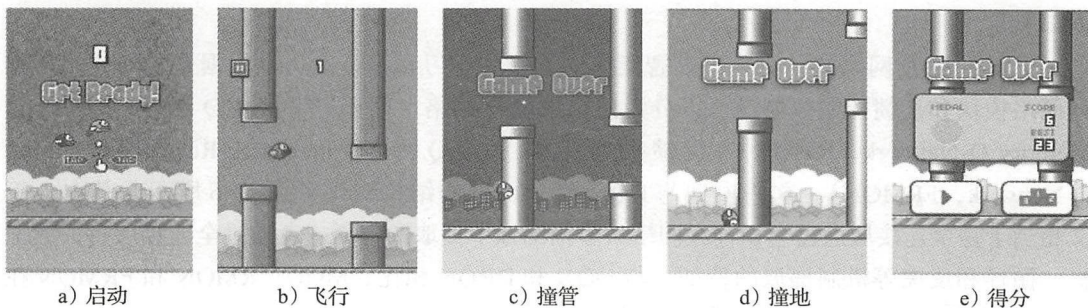


图 11.7 Flappy Bird 的典型界面

11.4.1 笨笨鸟网络的开发环境和工具包

笨笨鸟网络智能体是用 Python 语言开发的，其开发和运行依赖以下工具包：

- 1) Python 3.5.3 (或其他 Python 3.5.x 版本)
- 2) TensorFlow 1.1.0
- 3) Pygame 1.9.3 (一组用来开发游戏软件的 Python 程序模块)
- 4) Opencv-python 3.2.0
- 5) Pillow 4.1.1 (图像处理库，依赖 VC++ 2015)

Windows 下 Python 3.5.3 安装很简单，只需下载安装包后双击，选择好安装路径就可以直接安装，注意在安装完成后应把 Python 3.5.3 的安装路径拷贝到环境变量 Path 中。其他 4 个依赖工具包都是在 64 位的 Win10 操作系统上进行安装（Win7 上也可以），下载之前应查看一下计算机的“系统类型”以避免安装失败。这 4 个依赖工具包的安装过程如下：

- 1) 直接进入 pip3 所在目录（一般在 Python3.5.3 安装路径的子文件夹 Scripts 下）。
- 2) 执行命令：pip3 install tensorflow，如图 11.8 所示。

```

C:\Users\Administrator>cd AppData\Local\Programs\Python\Python35\Scripts
C:\Users\Administrator\AppData\Local\Programs\Python\Python35\Scripts>pip3 install tensorflow
Collecting tensorflow
  Downloading tensorflow-1.1.0-cp35-cp35m-win_and64.whl (19.4MB)
    100% |#####| 19.4MB 43kB/s
Collecting numpy>=1.11.0 (from tensorflow)
  Downloading numpy-1.13.0-cp35-none-win_and64.whl (7.8MB)
    100% |#####| 7.8MB 82kB/s
Collecting wheel>=0.26 (from tensorflow)
  Downloading wheel-0.29.0-py2.py3-none-any.whl (66kB)
    100% |#####| 71kB 139kB/s
Collecting protobuf>=3.2.0 (from tensorflow)
  Downloading protobuf-3.3.0.tar.gz (271kB)
    100% |#####| 276kB 124kB/s
Collecting six>=1.10.0 (from tensorflow)
  Downloading six-1.10.0-py2.py3-none-any.whl
Collecting werkzeug>=0.11.10 (from tensorflow)
  Downloading Werkzeug-0.12.2-py3-none-any.whl (312kB)
    100% |#####| 317kB 134kB/s
Requirement already satisfied: setuptools in c:\users\administrator\appdata\local\programs\python\python35\lib\site-packages (from protobuf>=3.2.0->tensorflow)
Installing collected packages: numpy, wheel, six, protobuf, werkzeug, tensorflow
  Running setup.py install for protobuf ... done
Successfully installed numpy-1.13.0 protobuf-3.3.0 six-1.10.0 tensorflow-1.1.0 werkzeug-0.12.2 wheel-0.29.0
C:\Users\Administrator\AppData\Local\Programs\Python\Python35\Scripts>
  
```

图 11.8 安装 TensorFlow 的过程

3) 执行命令: `pip3 install pygame-1.9.3-cp35-cp35m-win_amd64.whl`, 如图 11.9 所示。

```
C:\Users\Administrator\AppData\Local\Programs\Python\Python35\Scripts>pip3 install pygame-1.9.3-cp35-cp35m-win_amd64.whl
Processing c:\users\administrator\appdata\local\programs\python\python35\scripts\pygame-1.9.3-cp35-cp35m-win_amd64.whl
Installing collected packages: pygame
Successfully installed pygame-1.9.3

C:\Users\Administrator\AppData\Local\Programs\Python\Python35\Scripts>pip3 install opencv_python-3.2.0-cp35-cp35m-win_amd64.whl
Processing c:\users\administrator\appdata\local\programs\python\python35\scripts\opencv_python-3.2.0-cp35-cp35m-win_amd64.whl
Installing collected packages: opencv-python
Successfully installed opencv-python-3.2.0

C:\Users\Administrator\AppData\Local\Programs\Python\Python35\Scripts>
```

图 11.9 安装 pygame 和 opencv_python 的过程

4) 执行命令: `pip3 install opencv_python-3.2.0-cp35-cp35m-win_amd64.whl`, 如图 11.9 所示。

5) 执行命令: `pip3 install Pillow`, 如图 11.10 所示。

```
C:\Users\Administrator\AppData\Local\Programs\Python\Python35\Scripts>pip install Pillow
Collecting Pillow
  Downloading Pillow-4.1.1-cp35-cp35m-win_amd64.whl (1.5MB)
    100% |#####| 1.5MB 213kB/s
Collecting olefile (from Pillow)
  Downloading olefile-0.44.zip (74kB)
    100% |#####| 81kB 303kB/s
Building wheels for collected packages: olefile
  Running setup.py bdist_wheel for olefile ... done
  Stored in directory: C:\Users\Administrator\AppData\Local\pip\Cache\wheels\20\58\49\cc7bd00345397059149a10b0259ef38b867935ea2ecff99a9b
Successfully built olefile
Installing collected packages: olefile, Pillow
Successfully installed Pillow-4.1.1 olefile-0.44

C:\Users\Administrator\AppData\Local\Programs\Python\Python35\Scripts>
```

图 11.10 安装 Pillow 的过程

需要指出的是, Pygame 1.9.3 和 Opencv-python 3.2.0 必须提前下载, 它们的相应版本分别为 `pygame-1.9.3-cp35-cp35m-win_amd64.whl`、`opencv_python-3.2.0-cp35-cp35m-win_amd64.whl`。另外, 在不指定版本时安装 TensorFlow, 实际上安装的是最新版本, 不一定是 TensorFlow 1.1.0。如需指定版本, 可参考附录 C。

注意: 本案例也可在 Python 的集成环境 Anaconda 下运行, TensorFlow 的安装可参考附录 C。其他工具包的安装命令与之类似。

11.4.2 笨笨鸟网络的代码实现及说明

FBDQN 是深度强化学习的一个入门级案例, 可以代替人类玩家去控制小鸟的飞行状态, 目前互联网上已有多个版本, 不同之处一般只是重写或替换源工程文件的部分代码。FBDQN 源工程文件的一个下载链接为 <https://github.com/yenchennlin/DeepLearningFlappyBird>, 解压后的源工程文件名称为 `DeepLearningFlappyBird-master`, 能够在 Python3.5.3 版本下正

常运行，只要安装了所有依赖工具包。

注意，FBDQN 的有些版本只能够在较低的 Python 版本上运行，比如 Python2.7.x 系列版本。如果读者从其他链接下载 FBDQN，请仔细阅读工程中的 README.md 文件，查看所需软件配置。另外，为了便于管理和创建 Python 工程，建议读者下载 PyCharm 软件（收费，但有试用期）。在 PyCharm 安装成功之后，打开 DeepLearningFlappyBird-master 工程，如图 11.11 所示，可以看到 3 个扩展名为 .py 的文件：flapp_bird_utils.py、wrapped_flappy_bird.py 和 deep_q_network.py。其中，flapp_bird_utils.py 是一个工具类，主要用来加载 assets 目录下的所有资源文件。wrapped_flappy_bird.py 主要用来提供深度强化学习的游戏环境，可调用二维游戏引擎 Pygame 绘制窗口场景，并通过函数 frame_step(self, input_actions) 对笨笨鸟的动作进行评估，产生奖励回报、游戏状态和结束标志等信息。input_actions 代表笨笨鸟的飞行动作，即上升，还是下落。

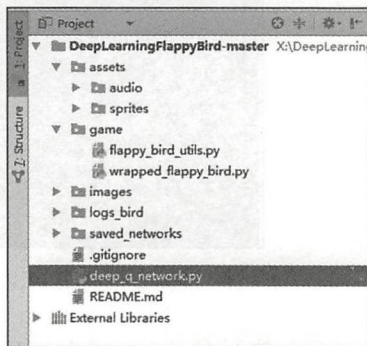


图 11.11 FBDQN 的源工程文件

在 DeepLearningFlappyBird-master 中，最重要的文件是 deep_q_network.py，用来描述深度强化学习网络智能体的结构。这个文件包含两个核心模块：卷积网络和强化学习。卷积网络模块的作用是根据游戏当前状态来预测当前应该执行的最佳动作，其中当前状态是用 4 帧连续的游戏窗口图像来表示的，最佳动作是指 Q 值最大的动作，整体网络结构如图 11.12 所示。强化学习模块的作用是对卷积网络的权值和偏置进行学习训练，使其实现一个从游戏当前状态到动作 Q 值的映射。在笨笨鸟游戏中，只有两个非常简单的动作：要么模拟人类游戏玩家点击屏幕，使笨笨鸟向上飞；要么什么也不做，使笨笨鸟以一定的速度下降。下面是这两个模块的主要实现代码及说明。

1. 卷积网络模块的实现代码以及说明

```
def weight_variable(shape):          # 初始化卷积网络的权值
    initial = tf.truncated_normal(shape, stddev = 0.01)
    return tf.Variable(initial)

def bias_variable(shape):            # 初始化卷积网络的偏置
    initial = tf.constant(0.01, shape = shape)
    return tf.Variable(initial)

def conv2d(x, W, stride):            # 将矩阵  $x$  与卷积核  $W$  进行步长为 stride 的卷积运算
    return tf.nn.conv2d(x, W, strides = [1, stride, stride, 1], padding = "SAME")

def max_pool_2x2(x):                # 对矩阵  $x$  进行最大池化操作
    return tf.nn.max_pool(x, ksize = [1, 2, 2, 1], strides = [1, 2, 2, 1], padding = "SAME")

def createNetwork():                # 创建卷积网络的结构
    W_conv1 = weight_variable([8, 8, 4, 32])          # 定义第 1 个卷积层的权值
    b_conv1 = bias_variable([32])                     # 定义第 1 个卷积层的偏置
    W_conv2 = weight_variable([4, 4, 32, 64])          # 定义第 2 个卷积层的权值
```

```

b_conv2 = bias_variable([64])           # 定义第 2 个卷积层的偏置
W_conv3 = weight_variable([3, 3, 64, 64]) # 定义第 3 个卷积层的权值
b_conv3 = bias_variable([64])           # 定义第 3 个卷积层的偏置
W_fc1 = weight_variable([1600, 512])     # 定义第 1 个全连接层的权值
b_fc1 = bias_variable([512])             # 定义第 1 个全连接层的偏置
W_fc2 = weight_variable([512, ACTIONS])  # 定义第 2 个全连接层的权值
b_fc2 = bias_variable([ACTIONS])         # 定义第 2 个全连接层的偏置

s = tf.placeholder("float", [None, 80, 80, 4]) # 定义输入层
h_conv1 = tf.nn.conv2d(s, W_conv1, [1, 1, 1, 1], b_conv1) # 计算第 1 个卷积层的输出
h_pool1 = tf.nn.max_pool(h_conv1, [1, 2, 2, 1], b_conv1) # 计算第 1 个池化层的输出
h_conv2 = tf.nn.conv2d(h_pool1, W_conv2, [1, 1, 1, 1], b_conv2) # 计算第 2 个卷积层的输出
h_conv3 = tf.nn.conv2d(h_conv2, W_conv3, [1, 1, 1, 1], b_conv3) # 计算第 3 个卷积层的输出
h_conv3_flat = tf.reshape(h_conv3, [1, 1600]) # 进行张量结构转换
h_fc1 = tf.nn.matmul(h_conv3_flat, W_fc1) + b_fc1 # 计算第 1 个全连接层的输出
readout = tf.matmul(h_fc1, W_fc2) + b_fc2 # 计算卷积网络的最后输出
return s, readout, h_fc1

```

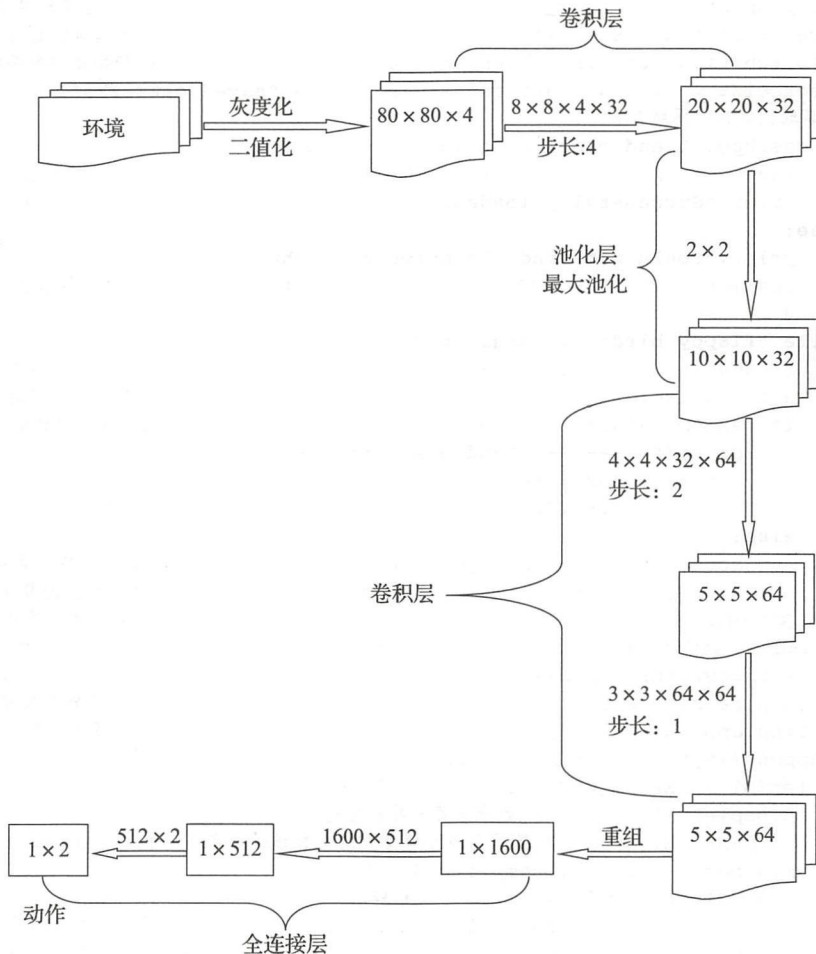


图 11.12 笨笨鸟案例的卷积网络结构

2. 强化学习模块的实现代码以及说明

```
def trainNetwork(s, readout, h_fc1, sess):                                # 进行强化学习的训练函数
    a=tf.placeholder("float", [None,ACTIONS])                          # 定义动作结构
    y=tf.placeholder("float", [None])                                    # 定义目标 Q 值结构
    readout_action=tf.reduce_sum(tf.multiply(readout,a),reduction_indices=1)
    # 计算动作 Q 值
    cost=tf.reduce_mean(tf.square(y - readout_action))                  # 计算动作 Q 值的均方误差
    train_step=tf.train.AdamOptimizer(1e-6).minimize(cost)              # 调用 Adam 优化算法
    game_state=game.GameState()                                         # 加载游戏环境
    D=deque()                                                            # 创建一个双向队列，即经验池
    do_nothing=np.zeros(ACTIONS)                                         # 初始化游戏动作数组
    do_nothing[0]=1                                                      # 定义什么都不做的动作
    x_t,r_0,terminal=game_state.frame_step(do_nothing)                  # 笨笨鸟执行动作后的返回值
    x_t=cv2.cvtColor(cv2.resize(x_t,(80,80)),cv2.COLOR_BGR2GRAY)
    # 对游戏当前界面归一化
    ret, x_t = cv2.threshold(x_t,1,255,cv2.THRESH_BINARY)                # 对归一化结果二值化
    s_t = np.stack((x_t, x_t, x_t, x_t), axis=2)                         # 将二值化结果转换成 4 通道状态
    saver = tf.train.Saver()                                              # 构造训练过程的存取对象
    sess.run(tf.global_variables_initializer())                          # 初始化所有参数
    checkpoint = tf.train.get_checkpoint_state("saved_networks")
    # 读取已保存的网络参数
    if checkpoint and checkpoint.model_checkpoint_path:
        saver.restore(sess, checkpoint.model_checkpoint_path) # 恢复已保存的网络参数
        print("Successfully loaded:", checkpoint.model_checkpoint_path)
    else:
        print("Could not find old network weights")
        epsilon = INITIAL_EPSILON                                       # 随机初始化所择动作的概率，缺省值为 0.0001
        t = 0
        while "flappy bird" != "angry bird":
            readout_t = readout.eval(feed_dict={s:[s_t]})[0] # 输入当前状态计算 Q 值
            a_t = np.zeros([ACTIONS])                                   # 初始化游戏动作数组
            if random.random() <= epsilon:                             # 以 epsilon 为概率随机选择动作
                print("-----Random Action-----")
                action_index = random.randrange(ACTIONS)
                a_t[random.randrange(ACTIONS)] = 1                     # 随机选择动作
            else:
                action_index = np.argmax(readout_t)                    # 计算最大 Q 值的索引
                a_t[action_index] = 1                                  # 根据最大 Q 值的索引找到相应的最优动作
            x_t1_colored,r_t,terminal=game_state.frame_step(a_t) # 笨笨鸟执行该动作后的返回值
            x_t1=cv2.cvtColor(cv2.resize(x_t1_colored,(80,80)),cv2.COLOR_BGR2GRAY) # 归一化
            ret,x_t1=cv2.threshold(x_t1, 1, 255, cv2.THRESH_BINARY)      # 二值化
            x_t1=np.reshape(x_t1,(80,80,1))                             # 把二值化结果转化成单通道数据
            s_t1=np.append(x_t1, s_t[:, :, :3],axis=2)                 # 生成新的 4 通道状态
            D.append((s_t,a_t,r_t,s_t1,terminal))                      # 扩充经验池
            if len(D) > REPLAY_MEMORY:                                  # 经验池大小超过 50 000
                D.popleft()                                             # 删除保留的最早经验样本
            if t > OBSERVE:                                             # 笨笨鸟与环境交互的次数超过预定观察值 OBSERVE=100 000
                minibatch = random.sample(D,BATCH)
                # 从经验池中随机选取 BATCH=32 个经验样本
                s_j_batch = [d[0] for d in minibatch]                  # 从 32 个经验样本读取当前状态 s_t
                a_batch = [d[1] for d in minibatch]                    # 读取 32 个当前状态的动作 a_t
                r_batch = [d[2] for d in minibatch]                    # 读取对 32 个动作的奖赏值 r_t
```



```

s_j1_batch = [d[3] for d in minibatch]    # 读取 32 个动作的相应后续状态
y_batch = []    # 定义目标 Q 值列表
readout_j1_batch=readout.eval(feed_dict={s:s_j1_batch})
# 计算后续状态的 Q 值
for i in range(0,len(minibatch)):
    terminal = minibatch[i][4]
    if terminal: # terminal=true    # 判断游戏是否结束
        y_batch.append(r_batch[i])    # 把结束时的动作奖赏值作为 Q 值
    else:
        y_batch.append(r_batch[i]+GAMMA*np.max(readout_j1_batch[i])) # 计算目标 Q 值
    train_step.run(feed_dict={
        # 运行梯度下降算法优化卷积网络的权值和偏置
        y:y_batch,
        a:a_batch,
        s:s_j_batch})
s_t = s_t1
t += 1
if t % 10000 == 0:    # 笨笨鸟与环境每交互 10 000 次就保存一次网络参数
    saver.save(sess, 'saved_networks/'+GAME+'-dqn', global_step=t)

```

11.4.3 笨笨鸟网络的学习训练过程

“笨笨鸟”的学习训练过程非常简单，具体说明如下。

1) 启动“笨笨鸟”游戏。将工程文件拷贝到 X 盘根目录：X:\DeepLearningFlappyBird-master。运行 deep_q_network.py 文件，命令为 python deep_q_network.py，如图 11.13 所示。图 11.14 是启动后的部分截图。其中，“TIMESTEP”是笨笨鸟与环境之间的交互次数，“STATE”是笨笨鸟与环境交互的阶段标志，并不是环境的状态。在 TIMESTEP 小于 100 000 时，STATE 的标志是“observe”，表示笨笨鸟对整个游戏的环境还“相当陌生”，需要不停地观察和探索。“EPSILON”是 ε 贪婪策略随机选择动作的概率 ε ，这里 $\varepsilon = 0.0001$ ，表示笨笨鸟以 0.0001 的概率随机选择一个动作，以 0.9999 的概率选择“最优”动作。“ACTION”是笨笨鸟采取的动作，取值为 0 或 1，取 0 表示玩家什么都不做，笨笨鸟向下飞翔（对应于卷积网络的输出 $[1, 0]$ ），1 表示玩家点击屏幕，笨笨鸟向上飞翔（对应于卷积网络的输出 $[0, 1]$ ）。“REWARD”表示环境对笨笨鸟选取动作的奖赏，在不同计分情况下的动作奖赏值列举在表 11.1 中。“Q_MAX”表示卷积神经网络对当前状态输出的最大 Q 值。

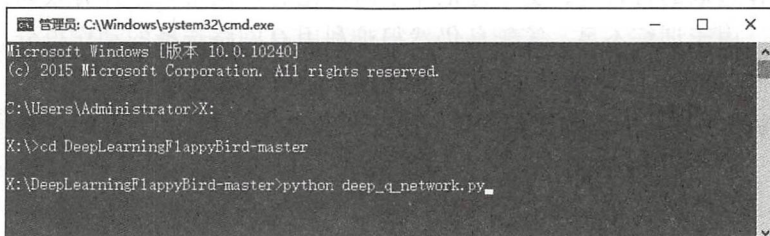


图 11.13 笨笨鸟游戏的启动

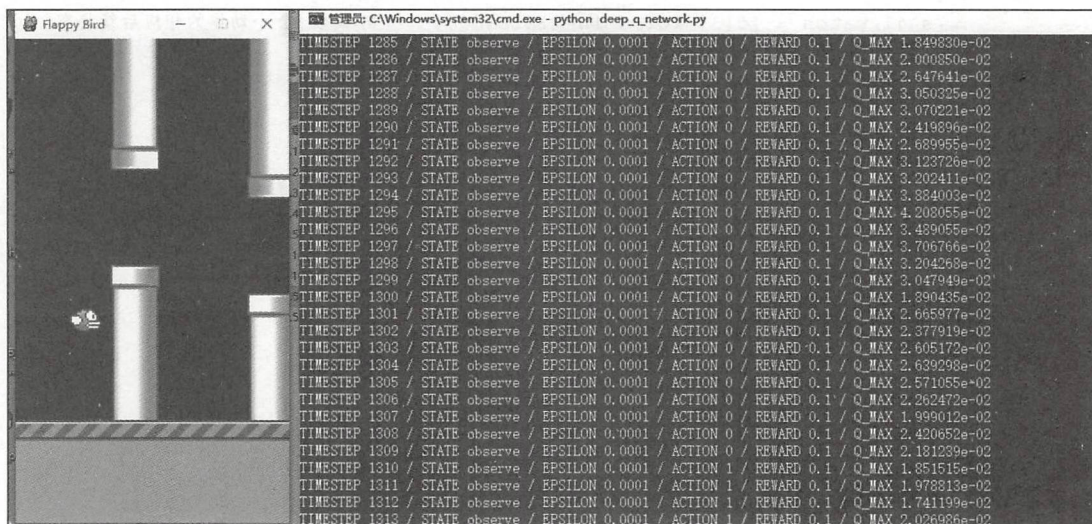


图 11.14 笨笨鸟游戏在启动后的部分截图

表 11.1 不同计分情况下的动作奖赏值

计分情况	动作奖赏值	
	玩家什么都不做	玩家点击屏幕
无碰撞、无穿越	0.1	0.1
穿越正对管道间隙	1	1
发生碰撞，游戏结束	-1	-1

2) 刚开始，笨笨鸟处于一种观察的状态，它选择动作的灵活性较差。例如，在图 11.14 中的 ACTION 这一列可以看到，笨笨鸟在第 1285 ~ 1309 步采取的动作都是向下飞翔，导致得分极为困难。事实上，笨笨鸟在 100000 步以前几乎 1 分都得不到。这是因为，笨笨鸟网络是在与环境交互 100000 次之后才真正开始使用强化学习训练和优化参数的。另外，笨笨鸟随机选择动作的概率很低，只有 0.0001。

3) 笨笨鸟在与环境交互 100000 次之后，开始进行强化学习训练，如图 11.15 所示。这时，笨笨鸟从观察阶段进入探索阶段，STATE 的标志是“explore”。在探索阶段，笨笨鸟将逐步减少动作选取的盲目性，更多地依靠不断优化的 Q 网络来选择有效动作。不过在探索阶段的初期，由于训练不足，笨笨鸟仍然很难利用 Q 网络选择的动作得分。比如，在训练 102197 次时，笨笨鸟还会撞到管道上，导致游戏结束，得分为 0。

4) 笨笨鸟的学习训练过程比较简单，首先把折扣因子取为 $\gamma = 0.99$ ，然后从经验池中随机选择 32 个样本 $e_{t_i} = (s_{t_i}, a_{t_i}, r_{t_i}, s_{t_{i+1}})$ 估计 32 个 Q 值 $Q(s_{t_i}, a_{t_i})$ ($i = 1, 2, \dots, 32$)，估计公式如下：

$$Q(s_t, a_t) \leftarrow r_{t+1} + 0.99 \times \max_{a'} Q(s_{t+1}, a') \quad (11.12)$$

其中,最后 32 个 Q 值 $Q(s_t, a_t)$ 用来近似计算梯度 $\nabla_{\theta} L(\theta)$, 并根据梯度下降算法更新网络参数 θ 。

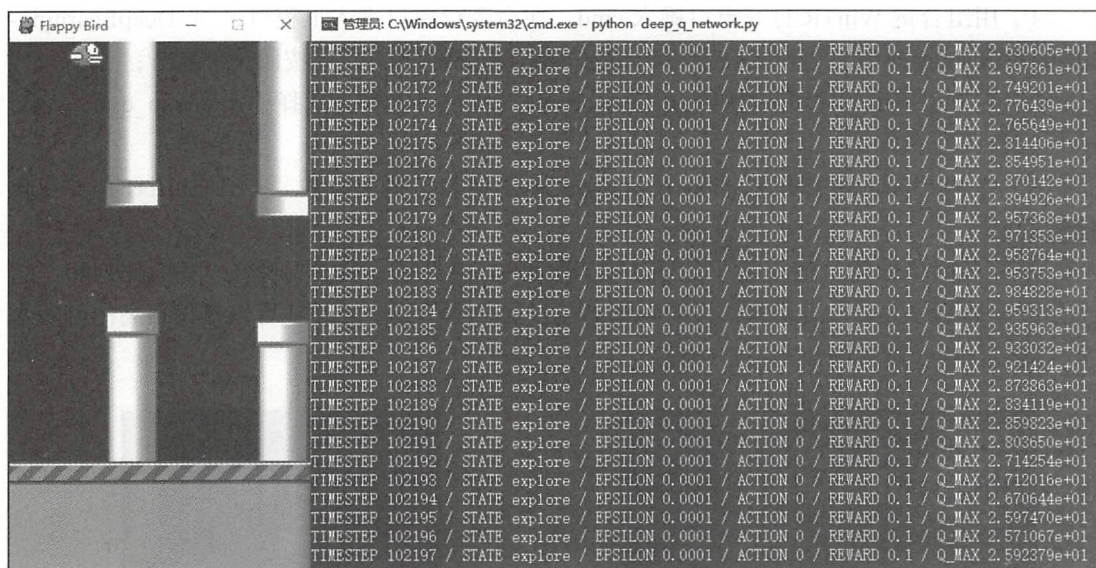


图 11.15 笨笨鸟游戏在训练过程的部分截图

5) 笨笨鸟经过 200 万~300 万次(只使用 CPU 约 30 小时)的训练,网络参数才能达到较优的取值。之后,笨笨鸟选择动作的智能水平很快提高,直到成为游戏高手,如图 11.16 所示。网络参数值保存在 savedNetwork 文件夹下以“bird-dqn”为前缀名的文件中。

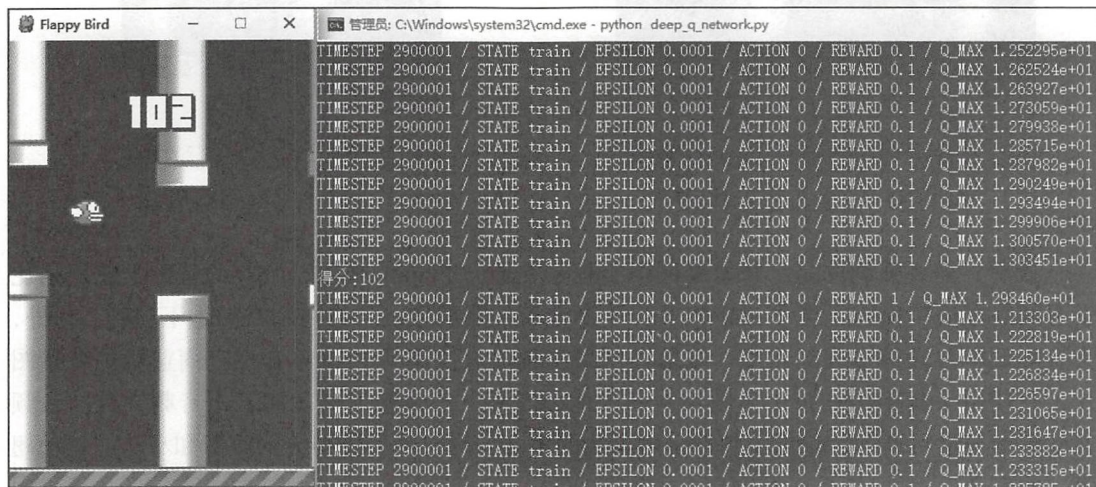


图 11.16 笨笨鸟游戏在训练完成后的运行结果

11.4.4 笨笨鸟网络的演示效果

笨笨鸟网络工程的核心文件是 `deep_q_network.py`，有下面 3 种运行方式：

- 1) 用组合键 Win+R 打开窗口键入 `cmd`，在命令行窗口进入工程目录 `X:\DeepLearning-FlappyBird-master` 后，再键入命令 “`Python deep_q_network.py`”，按回车键运行；
- 2) 在 PyCharm 中打开 “`deep_q_network.py`” 文件，右键，选择 “`run deep_q_network.py`” 运行；
- 3) 在 PyCharm 中打开 “`deep_q_network.py`” 文件，同时按住 “`Ctrl+Shift+F10`” 三个键运行。

笨笨鸟网络的运行演示效果如图 11.17 和图 11.18 所示，得分分别为 1385 和 1449。这个分数远远超过大量人类玩家，说明笨笨鸟经过强化学习训练后确实可以获得很高的游戏水平，能够玩很长的时间而不发生碰撞。

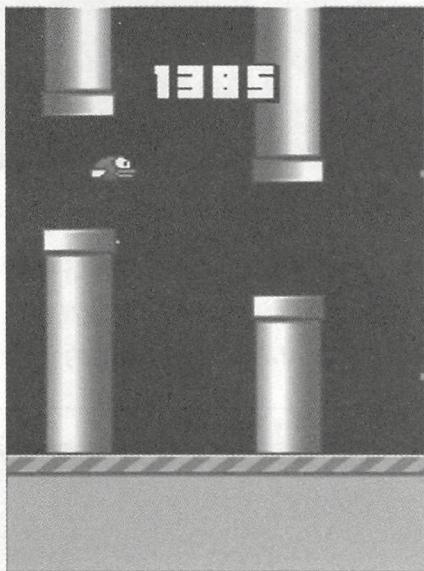


图 11.17 计算机得分 1385

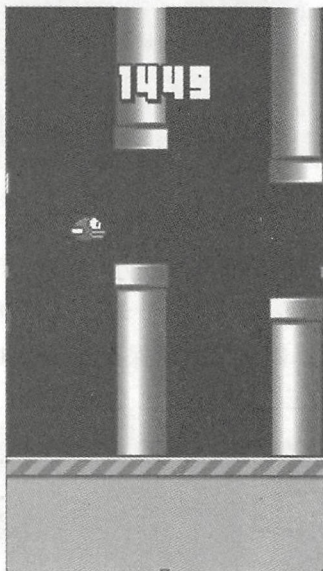


图 11.18 计算机得分 1449

最后，需要强调的是，笨笨鸟网络工程是在原 “Flappy Bird” 游戏的基础上使用强化学习网络方法实现的，与 “Flappy Bird” 商业版使用的语言、技巧和目的是不同的。另外，为了降低学习训练过程的难度，笨笨鸟网络工程没有使用 “Flappy Bird” 商业版的背景图片，只使用了其中的三张小鸟图片、一张地面图片、一张管道图片和若干音频文件。音频文件不参与学习训练，包括小鸟撞击障碍物的声音、穿过障碍物得分的声音和小鸟 “拍动翅膀” 的声音。

卷积神经网络的顶尖成就——AlphaGo

AlphaGo 是第一款击败人类职业世界冠军的人工智能围棋程序，由谷歌旗下 DeepMind 公司戴密斯·哈萨比斯领衔的团队开发。其主要工作原理是“深度学习”，特别是深层卷积神经网络与强化学习结合训练产生的估值网络（value network）和策略网络（policy network），及其对蒙特卡罗搜索树带来的性能提升。围棋界公认 AlphaGo 的棋力已经超过人类职业围棋顶尖水平，但前三个版本 AlphaGo Fan、AlphaGo Lee 和 AlphaGo Master 都需要用大量人类棋谱进行训练。而另一个新版本 AlphaGo Zero 只需采用基本规则进行自我强化学习，不再需要人类棋谱数据进行训练，这就大幅超越了 AlphaGo 之前版本的棋力水平。本章首先对人工智能棋类程序进行简单介绍，然后分析 AlphaGo 和 AlphaGo Zero 的设计原理，最后详述一个与 AlphaGo 原理类似的围棋程序 MuGo。

12.1 人工智能棋类程序简介

人工智能，就是使用计算机对人类智能的模仿，“学会”人类在某一领域的专业技能。早在人工智能发展初期的 20 世纪 50 年代，来自 IBM 工程研究组的 Samuel 就利用机器学习的思想开发出了跳棋程序。这个跳棋程序具有初步的学习能力，可以在与人对弈的过程中不断地积累经验，提高自己的棋艺，并且在 1956 年战胜了康涅狄格州的西洋跳棋冠军。跳棋程序的成功使得早期的人工智能学者大多认为“计算机很快就会战胜人类”，1957 年西蒙甚至乐观地预言：10 年内数字计算机将取代人类获得国际象棋世界冠军。

然而经过深入的研究，人们却发现人工智能所遇到的困难比想象中的多得多，比如，Samuel 的跳棋程序在击败州冠军后无法再前进。而在国际象棋对弈中，世界级大师 David Levy 与人工智能学者打了一个非常著名的赌：没有计算机国际象棋程序可以在十年内击败我，赌金为 1250 英镑。在这十年中，David Levy 成功击败了所有计算机挑战者，并在 1978 年 9 月举行的一场六局对决中，以 4.5 : 1.5 的比分战胜当时的终极计算机程序 Chess 4.7，最终赢得了这个赌。尽管赢下了赌注，计算机却在第四局对局中获得了胜利，这是计

算机程序历史上第一次击败人类国际象棋大师。最著名的“人机大战”是1997年俄罗斯国际特级大师卡斯帕罗夫与IBM公司研发的超级计算机深蓝（Deep Blue）对决。深蓝以深度优先树搜索技术为核心，结合手工设计的评价函数和 α - β 剪枝的启发式策略，不仅击败了人类最强的国际象棋特级大师（比分为3.5:2.5），也进一步激发了人们对人工智能围棋程序战胜人类顶尖棋手的渴望。

在人工智能棋类游戏中，树搜索是一种常用的技术。棋类活动每步一般都有好几种走法，可以把所有的走法描述成一棵“树”的形状，相应的术语就称为“搜索树”（search tree）。从搜索树中发现好棋的过程，就称为“树搜索”（tree searching）。简单的棋类，如“井字棋”“五子棋”，可以把所有的走法都找出来，包括对手的回应对走法，这样就能够形成比较正确的走法。因此，人们在下棋时，按照树搜索想得越深、越广，就越容易赢棋。结合 α - β 剪枝策略，精心设计的树搜索算法可以战胜人类的国际象棋冠军。不过，在国际象棋上取得大捷的搜索树技术，要想用在更加复杂的围棋上击败人类的顶级棋手，却又难上加难。围棋的标准棋盘为 19×19 大小，共361个落子点，理论上的棋局可能多达 2^{361} 种。这已足以表明围棋棋局的复杂程度远超国际象棋（棋盘只有 8×8 大小，32颗棋子，棋局至多 $64^{32} = 2^{192}$ 种）。所以，搜索树技术在围棋程序中的推广很难达到国际象棋程序的水平。正因为如此，创建高水平的围棋程序长期以来被称为人工智能领域的重大挑战。

顶尖的围棋程序AlphaGo，需要用到另一种搜索技术——蒙特卡罗树搜索。用蒙特卡罗树搜索虽然不能保证找到最佳的走法，但能够选择胜率较大的走法。最早的围棋程序是1968年Albert Lindsey Zobrist开发的，他引入了一个评估函数对棋局进行分析，来估算双方占空的大小，但难以达到专业棋手的水准。自2006年以来，随着蒙特卡罗树搜索和机器学习在围棋上的应用，很多围棋程序，包括疯石围棋（CrazyStone）、银星围棋（SilverStar）、天顶围棋（ZEN）等，在棋力水平上开始有了突飞猛进的增长，普遍提升到业余高段的水准。2011年8月欧洲围棋大会，ZEN在19路棋盘上被让五子击败日本职业棋手林耕三六段。2012年3月，ZEN被让四子击败了日本超一流棋手武宫正树九段，这是围棋程序首次在让四子的情况下战胜一流职业选手。2013年，CrazyStone被让四子击败日本石田芳夫九段，2014年被让四子击败日本依田纪基九段。2015年10月，Google旗下人工智能公司DeepMind将深度学习与蒙特卡罗树搜索相结合开发的AlphaGo，在没有任何让子的情况下，以五战全胜的成绩击败了欧洲围棋冠军、职业围棋二段樊麾，这也是计算机围棋程序首次击败围棋职业棋手。2016年3月，AlphaGo又以4:1战胜了人类的顶尖高手、世界围棋冠军、职业九段选手李世石（或芈）。2016年末2017年初，AlphaGo在中国棋类网站上以Master为注册账号与中日韩数十位围棋高手进行快棋对决，连续60局无一败绩。2017年5月，在中国乌镇围棋峰会上，AlphaGo与排名世界第一的围棋冠军柯洁对战，以3:0的总比分获胜，其研发团队宣布之后AlphaGo将不再参加围棋比赛。

AlphaGo的上述3个版本被DeepMind团队分别称为AlphaGo Fan、AlphaGo Lee和AlphaGo Master。其中，AlphaGo Fan和AlphaGo Lee都利用策略网络和估值网络对棋局

进行评估和落子选择，这些都需要用大量的人类棋谱进行训练。而随后出现的 AlphaGo Zero，虽然与 AlphaGo Master 使用相同的网络结构和学习算法，但棋力却获得了大幅提升。AlphaGo Zero 是目前最优秀的人工智能围棋，它将策略网络和估值网络合二为一，使用单一网络就能够实现落子位置的选择和棋盘状态的评估，除了基本规则之外无需领域知识，不再需要人类棋谱数据进行训练，也不再执行任何蒙特卡罗展开。AlphaGo Zero 可以依靠自弈强化学习从“零”开始不断地提升棋力，仅仅学习 3 天就能够以 100:0 的成绩完胜 AlphaGo Lee，学习 40 天就可以对战 AlphaGo Master。AlphaGo Zero 与最强版本的 AlphaGo Master 对战 100 局，每一局的时间都控制在 2 小时内，结果比分为 89:11。

下面，先分析 AlphaGo 和 AlphaGo Zero 的设计原理，再讨论仿效围棋程序 MuGo。

12.2 AlphaGo 的设计原理

围棋是一种完全信息博弈，有一个最优值函数，在每一个棋盘局面或状态 s ，都决定着所有玩家在完美发挥下的最终结局。理论上，这个最优值函数是可以通过搜索树递归计算的。搜索树的规模大约是 b^d ，其中 b 是博弈的宽度（即每个棋局的合法走步数）， d 是博弈的深度（或长度），是对所有可能走步序列数目的近似估计。在大规模博弈游戏中，比如象棋，特别是围棋，进行完全穷尽搜索是不可能的，但是可以通过两种启发式策略减小有效的搜索空间。一是通过状态评估减小深度，二是通过走步采样减小搜索宽度。状态评估可以通过一个近似估值函数对搜索树在棋局状态 s 之下的子树进行裁减来实现。走步采样可以通过在状态 s 下的策略，也就是可能走步的概率分布，针对两个玩家进行无分支蒙特卡罗最大展开搜索来实现，其中展开相当于一个对棋局试下的过程，最大展开就是把棋局一直试下到结束。近似估值函数的方法已经在国际象棋（chess）、西洋跳棋（checker）和奥赛罗棋（othello）等棋类游戏上获得了超人表现，但在复杂度更高的围棋上一度被认为不可能的。对蒙特卡罗展开求平均的方法则能够提供一种有效的状态评估，在西洋双陆棋（backgammon）和拼字游戏（scrabble）上达到超人表现，在围棋博弈上也可以达到低段业余水平。以蒙特卡罗展开为基础的蒙特卡罗树搜索（Monte Carlo Tree Search, MCTS），通过预测人类专业选手的走步策略，在围棋博弈上能够达到更高段位的业余水平。AlphaGo 通过把深层神经网络与蒙特卡罗树搜索相结合，使人工智能围棋达到了远远超过了人类顶尖高手的水平。下面将讨论 AlphaGo 的总体思路、训练流程和搜索过程。

12.2.1 总体思路

AlphaGo 的总体思路在于利用深层卷积神经网络来减小搜索树的有效深度和宽度：构造一个估值网络来评估状态，构造一个策略网络来采样走步。作为一个人工智能程序，AlphaGo 在本质上就是策略网络、估值网络和蒙特卡罗树搜索的有机结合。估值网络，就是用“估值”数来评估当前的棋局。如果把棋局上所有棋子的位置总和称为一个“状态”，每个状态可能允许若干不同的后续状态。所有可能状态的前后关系就构成了所谓的搜

索树。一个暴力的搜索算法会遍历这个搜索树的每一个子树。但是，有些状态是较容易判断输赢的，把这些状态用估值表示，就可以据此省略了对它所有后续状态的探索，即利用估值网络削减搜索深度。策略网络，就是指在给定棋局状态，评估每一种应对可能的胜率，从而根据当前盘面状态来选择走步策略。在数学上，就是估计一个在各个合理位置上下子获胜的可能的概率分布。因为有些下法的获胜概率很低，可忽略，所以通过策略网络可以消减搜索树的宽度。通俗地说，“估值”就是能看懂棋局，一眼就能判断某给定棋局是不是能赢，这是个偏宏观的评估。“策略”是指在每一步博弈时，各种选择的取舍，这是个偏微观的评估。AlphaGo 利用模拟棋手、强化自我的方法，在宏观和微观两个方面提高了探索的效率。

AlphaGo 对神经网络的训练过程是一条流水线，如图 12.1 所示，可以分解为三个机器学习的阶段。首先，直接利用围棋棋局训练一个有监督学习策略网络（supervised learning policy network），同时还训练一个快速走步策略网络（fast rollout policy network）。然后，通过优化自弈过程训练一个强化学习策略网络（reinforcement Learning network），以改进有监督学习策略网络，使策略调向赢棋的准确目标，而不是单纯地最大化预测准确率。最后，训练一个估值网络，以预测强化学习策略网络进行自弈的赢家。

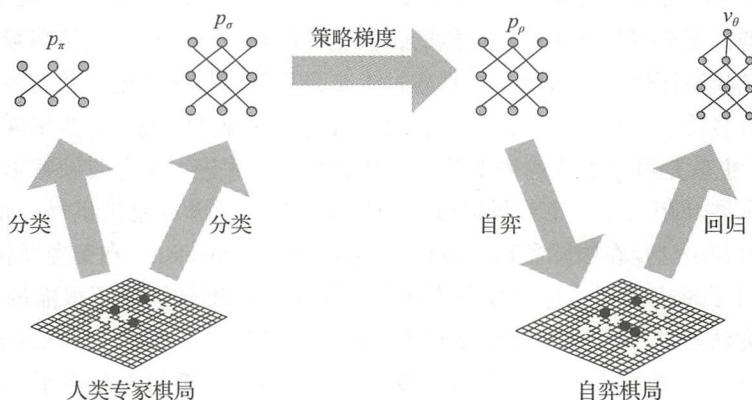


图 12.1 AlphaGo 的训练流程。 p_π 是快速走步策略网络， p_σ 是有监督学习策略网络， p_ρ 是强化学习策略网络， v_θ 是估值网络。和在人类专家棋局上训练，用于预测下一个走步。 p_ρ 先初始化为 p_σ ，再用策略梯度学习改进，以最大化自弈的胜率。估值网络 v_θ 进行回归训练以预测自弈棋局的期望胜率

12.2.2 训练流程

1. 策略网络的有监督学习训练

策略网络是一个 13 层的全卷积神经网络，输入是棋盘状态 s ，输出是 19×19 个概率值（棋盘是 19×19 的方格），对应下一步落子位置 a 的概率为 $p(a|s)$ 。实际上，AlphaGo 一共训练了两个策略网络：有监督学习策略网络 $p_\sigma(a|s)$ 和强化学习策略网络 $p_\rho(a|s)$ 。

$p_{\sigma}(a|s)$ 采用人类棋谱做训练数据, $p_{\rho}(a|s)$ 是在 $p_{\sigma}(a|s)$ 的基础上通过自弈的强化学习进一步更新参数, 提高性能。

训练流程的第一阶段是对策略网络 $p_{\sigma}(a|s)$ 进行有监督学习训练, 其中 σ 是权值。在 $p_{\sigma}(a|s)$ 中, 棋盘状态 s 是用 $19 \times 19 \times 48$ 的数据结构来表示的, 由 48 个 19×19 的特征面组成。第 1 个隐含层把输入用 0 填充成 23×23 大小的图像, 然后与大小为 5×5 、步长为 1 的 k 个滤波器 (即卷积核) 做卷积, 并且经过一次校正线性单元处理。在后续的 $2 \sim 12$ 个隐含层中, 每一个都把前面相应的卷积层用 0 填充成 21×21 的图像, 然后与大小为 3×3 、步长为 1 的 k 个滤波器做卷积, 并且再经过一次校正线性单元处理。最后一层与大小为 1×1 、步长为 1 的 1 个滤波器做卷积, 在每个位置再加上一个不同的偏置, 再通过一个软最大函数计算输出。滤波器的 k 个数一般取 128、192、256 和 384, 与樊麾比赛的版本取 $k = 192$ 。

$p_{\sigma}(a|s)$ 的训练是基于 KGS 围棋服务器上的 3 千万棋局通过随机采样状态走步对 (s, a) 来实现的, 方法是利用随机梯度上升最大化人类在棋盘状态 s 下选择走步 a 的可能性 (或似然函数), 即

$$\Delta \sigma \propto \frac{\partial \log p_{\sigma}(a|s)}{\partial \sigma} \quad (12.1)$$

如果仅用棋局状态和走步历史训练, $p_{\sigma}(a|s)$ 在测试集上的预测成功率是 55.7%。如果再加上其他特征, 预测成功率可以进一步提高到 57.0%。

另外, AlphaGo 也训练了一个快速走步策略网络 p_{π} , 以便在展开时快速采样。 p_{π} 采用线性软最大函数 (π 是权值), 输入较小的模式特征, 精度只能到达 24.2%, 但速度更快, 选择一次走步仅需 $2\mu\text{s}$, 而 p_{σ} 选择一次走步需要 3ms 。

2. 策略网络的强化学习训练

训练流程的第二阶段是对策略网络 $p_{\rho}(a|s)$ 进行强化学习, 其中 ρ 是权值。 $p_{\rho}(a|s)$ 是用 $p_{\sigma}(a|s)$ 来初始化的, 也就是说 $\rho = \sigma$, 因此它们具有相同的结构, 如图 12.2a 所示。 $p_{\rho}(a|s)$ 也利用梯度下降算法学习, 区别在于用一个“回报”来奖励那些会导致最终获胜的策略。 $p_{\rho}(a|s)$ 的目标是通过策略梯度强化学习改进 $p_{\sigma}(a|s)$ 的性能, 以最大化自弈的胜率。自弈是指用当前的 $p_{\rho}(a|s)$ 与之前某个随机选择的 $p_{\rho'}(a|s)$ 进行对弈。按照这种方式随机选择对手 $p_{\rho'}(a|s)$, 可以稳定训练过程, 防止对当前策略的过拟合。同时, 强化学习还需要一个奖赏函数 $r(s)$ 。这个奖赏函数在所有非结局时刻取 0 值。用 $z_t = \pm r(s_t)$ 表示从第 t 时刻的当前棋手看所得到的结局奖赏: 赢棋取 +1, 输棋取 -1。权值 ρ 在每个时刻 t 都采用随机梯度上升沿着最大化期望奖赏的方向更新, 即

$$\Delta \rho \propto \frac{\partial \log p_{\rho}(a_t|s_t)}{\partial \rho} z_t \quad (12.2)$$

$p_{\rho}(a|s)$ 可以看作对 $p_{\sigma}(a|s)$ 的加强, 通过 $a_t \sim p_{\rho}(\cdot|s_t)$ 选择落点走棋的能力得到了很大提高, 测试表明对 $p_{\sigma}(a|s)$ 的胜率可达 80% 多。而对最强的开源围棋程序 Pachi,



$p_\rho(a|s)$ 不用任何搜索, 胜率就可以达到 85%。之前最先进的有监督学习卷积神经网络, 对 Pachi 的胜率是 11%, 对稍弱程序 Fuego 的胜率是 12%。

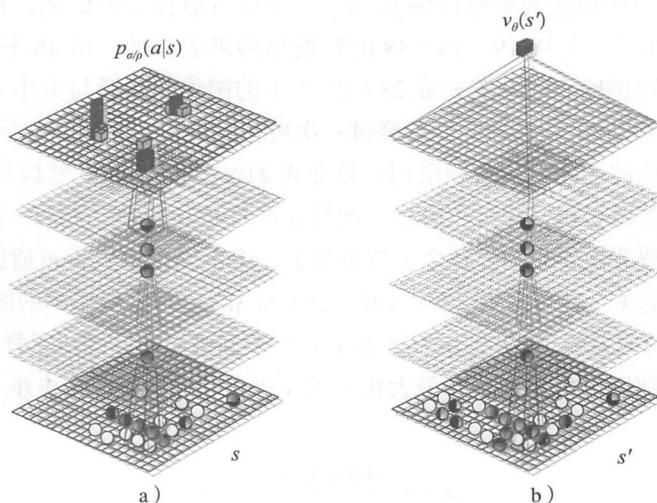


图 12.2 AlphaGo 的神经网络结构示意图: a) 策略网络或 b) 估值网络。策略网络输入的是棋局的状态 s , 让其经过许多卷积层的处理 (p_σ 的权值为 σ , p_ρ 的权值为 ρ), 输出下一个合法走步的概率, 用棋盘上的概率图表示。估值网络也采用许多卷积层 (权值为 θ), 但输出是一个标量, 用于预测棋局的期望胜率

3. 估值网络的强化学习训练

训练流程的最后阶段关注于棋局评估, 目的是设计一个估值函数 $v^\rho(s)$ 预测从棋局状态 s 开始博弈双方都按照策略 p 下棋的结局, 即

$$v^\rho(s) = E[z_t | s_t = s, a_{1:T} \sim p] \quad (12.3)$$

理想情况下, 应该是采用完美发挥下的最优估值函数 $v^*(s)$ 。实际上, 只能使用替代策略, 比如强化学习策略网络 p_ρ 去估计最好的估值函数 $v^{\rho\rho}(s)$ 。这个估值函数可以用一个具有权值 θ 的估值网络 $v_\theta(s)$ 来近似: $v_\theta(s) \approx v^{\rho\rho}(s) \approx v^*(s)$ 。

估值网络与策略网络的结构类似, 但输出是一个单值预测结局, 而不是一个概率分布。估值网络的结构如图 12.2b 所示, 其输入也是 $19 \times 19 \times 48$ 的图像堆叠结构, 另加一个描述当前走子方的二值特征面。从第 2 ~ 11 个隐含层, 估值网络与策略网络完全一样, 第 12 个隐含层是一个多加的卷积层, 第 13 个隐含层用大小为 1×1 、步长为 1 的 1 个滤波器做卷积, 第 14 个隐含层是一个具有 256 个校正线性单元的全连接线性层。估值网络的输出是一个全连接线性层, 只有一个类型为 \tanh 的非线性激活单元。估值网络是一个回归模型, 用随机梯度下降算法在状态结局对 (s, z) 上训练,

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial \theta} (z - v_\theta(s)) \quad (12.4)$$



其目标是最小化预测值 $v_\theta(s)$ 和期望值 z 之间的最小均方误差 (Mean Squared Error, MSE)。

用整盘对局数据预测结局的朴素方法会导致过拟合。这是因为相继棋局状态是高度相关的, 只差一个棋子, 但回归目标在整盘对局中都是相同的。估值网络在 KGS 数据集上如此训练只是能记住对弈的结局而不是推广到新的棋局状态, 训练 MSE 为 0.19, 测试 MSE 为 0.37。缓解这个问题的方法是, 生成一个新的自弈数据集, 包含 3 千万不同的棋局, 其中每一个棋局都从不同的独立对弈中采样得到, 而每一次对弈都是在强化学习网络 p_ρ 及其自身之间进行, 直到结束。在这个数据集上训练, 得到的训练 MSE 是 0.226, 测试 MSE 是 0.234, 说明过拟合问题已经很小。对比实验表明, 估值网络 $v_\theta(s)$ 对棋局评估的准确率一致优于采用快速走步策略网络 p_π 的蒙特卡罗展开。而且, 估值网络仅用一次评估, 准确率就已经接近采用强化学习策略网络 p_ρ 的蒙特卡罗展开, 但计算量却是后者的 1/15 000。

4. 训练过程小结

AlphaGo 一共训练了四个神经网络: 快速走步策略网络 p_π 、有监督学习策略网络 p_σ 、强化学习策略网络 p_ρ 和估值网络 v_θ 。前三个网络 (p_π 、 p_σ 和 p_ρ) 都输入当前棋局状态, 输出是棋盘上各点走子的概率。 p_π 是一个结构相对简单的网络, 水平低但计算量小。 p_σ 的网络结构更为复杂, 水平高但计算量大。 p_ρ 是 p_σ 的加强版, 用 p_σ 初始化, 与 p_σ 的结构相同、计算量相同, 但水平更高。第四个网络 v_θ 是一种回归模型, 也输入当前棋局状态, 但输出是对弈双方的胜率。

最后, p_π 和 p_σ 都是用人类专家对弈棋局通过有监督学习训练的, 而 p_ρ 和 v_θ 都是用自弈棋局通过强化学习训练的。这些网络之间的关系如图 12.1 和图 12.2 所示。

12.2.3 搜索过程

在深层神经网络的训练完成之后, AlphaGo 就可以开始与人对弈了。在对弈过程中, AlphaGo 的“思考”是通过蒙特卡罗博弈树搜索和模拟来实现的。所谓搜索, 就是给定一个棋局, 确定如何走下一步, 分为“向前看”和“回头看”两个环节。向前看就是选择最有可能获胜的走步, 直到能分出胜负的“叶子”节点。回头看就是根据棋局的演化结果, 回溯更新搜索树对棋局的评估。

AlphaGo 在蒙特卡罗树搜索算法中集成了策略网络和估值网络 (如图 12.3 所示), 通过向前看选择走步。搜索树的每条边 (s, a) 存储了走步估值 $Q(s, a)$ 、访问次数 $N(s, a)$ 和先验概率 $P(s, a)$ 。模拟过程从树根状态开始, 一直穿越到叶节点 (即在整个对弈过程中无回溯地沿树下行)。在每个时刻 t , 模拟都根据棋局状态 s 选择下一个走步 a_t , 即

$$a_t = \arg \max_a (Q(s_t, a) + u(s_t, a)) \quad (12.5)$$

用来最大化走步估值加额外奖励 $u(s, a)$ 。其中, $u(s, a)$ 正比于先验概率, 但随访问次数增加而变小以利于各种不同情况的探索, 即

$$u(s, a) \propto \frac{P(s, a)}{1 + N(s, a)} \quad (12.6)$$



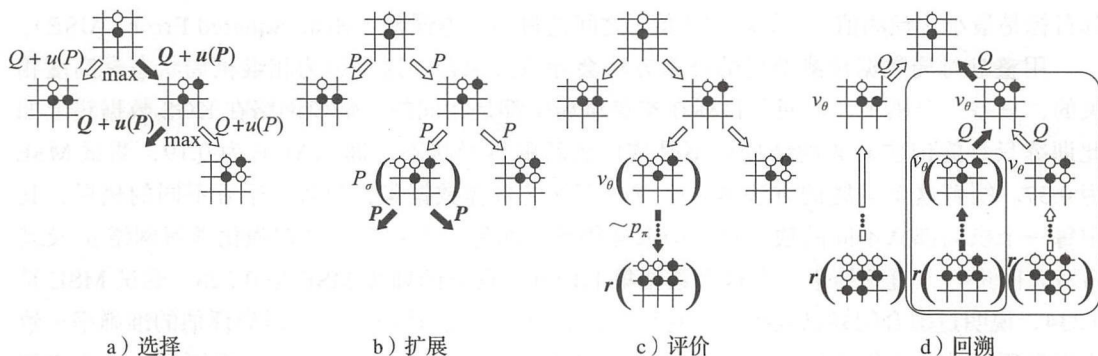


图 12.3 AlphaGo 的蒙特卡罗树搜索过程。每一次模拟在穿过搜索树时选择的边，都具有最大的“走步值 Q 加奖励值 $u(P)$ ”，其中 $u(P)$ 依赖于该边存储的先验概率。叶节点可以扩展，每个走步产生的新节点用策略网络 p_σ 计算输出概率，结果存储在先验概率 P 中。在模拟结束时，叶节点用两种方式评价：估值网络 v_θ ，或者用快速走步策略网络 p_π 把棋局展开到结束，计算奖励函数 r 。更新走步值 Q ，以便在下面的子树中追溯所有评价的均值 $r(\cdot)$ 和 $v_\theta(\cdot)$

当穿越过程在第 L 步达到一个叶节点棋局 s_L 时，该叶节点可能被扩展。叶节点棋局 s_L 只会被有监督策略网络 p_σ 处理一次。用 p_σ 计算合法走步概率并存储在先验概率中，即 $P(s, a) = p_\sigma(a | s)$ 。评估叶节点常用两种方法：一是通过估值网络 $v_\theta(s_L)$ 计算，二是用快速走步策略网络 p_π 产生一个直到结局时刻 T 的随机走步序列，通过其输赢结果 z_L 来计算。另一种方法是引入一个混合参数 λ 把这两种方法结合起来：

$$V(s_L) = (1 - \lambda) v_\theta(s_L) + \lambda z_L \quad (12.7)$$

在模拟结束后，所有被穿越边的走步估值和访问次数都将被更新。每条边都累加在所有模拟中通过该边的访问次数和平均走步估值。

$$N(s, a) = \sum_{i=1}^n 1(s, a, i) \quad (12.8)$$

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^n 1(s, a, i) V(s_L^i) \quad (12.9)$$

其中， s_L^i 是第 i 次模拟的叶节点， $1(s, a, i)$ 表示边 (s, a) 在第 i 次模拟期间是否被穿过。一旦搜索完成，算法就从树根节点开始，选择被访问最多的边，即用得最多的走步（或走法、着法）。与最大化走步值相比，这种最大化访问次数的方法效果更稳定，对异常情况不敏感。如果通过最大化访问次数选择的走步与通过最大化走步值选择的不同，AlphaGo 的竞赛版在对手走棋时还会继续搜索，但时间受中局规定的限制。在总体评分低于胜率的 10% 以下时（相当于 $\max_a Q(s, a) < -0.8$ ），AlphaGo 会投子弃赛认输。

值得一提的是，AlphaGo 的有监督学习策略网络 p_σ 在性能上优于更强大的强化学习策略网络 p_ρ ，原因可能是人类喜欢选择多样性的潜力走法，而 p_ρ 只优化单一的最好走法。不过，用 p_ρ 训练得到的估值函数 $v_\theta(s) \approx v^{\rho}(s)$ 又要比用 p_σ 训练得到的估值函数 $v_\theta(s) \approx v^{\sigma}(s)$



性能更优。

与传统搜索启发式技术相比,利用策略网络和估值网络提高 MCTS 的性能,需要的计算量增加了好几个数量级。为了高效集成较大的神经网络,AlphaGo 实现了一种异步策略估值 MTCS 算法 (APV-MCTS)。在 APV-MCTS 中,搜索树的每个节点 s 都包含所有合法走步的边 $(s, a), a \in A(s)$, 而每条边还需要存储很多统计特征, 即 $\{P(s, a), N_v(s, a), N_r(s, a), W_v(s, a), W_r(s, a), Q(s, a)\}$ 。其中, $P(s, a)$ 是先验概率; $W_v(s, a)$ 和 $W_r(s, a)$ 是总体走步估值的蒙特卡罗估计, 分别在叶节点评价 $N_v(s, a)$ 和展开奖励 $N_r(s, a)$ 上进行累积; $Q(s, a)$ 是关于边 (s, a) 的组合平均走步估值。这些特征需要在独立的搜索线程上进行大量模拟来计算。如图 12.3 所示, APV-MCTS 算法的运行可以分解为下面四个阶段。

1. 选择 (selection)

在每次模拟中, 第一个树内阶段都从搜索树根开始, 经过 L 步后达到叶节点时结束。对 $t < L$ 的每一步, 都根据搜索树的统计数据, 利用 PUCT 算法的变种来选择走步, 即

$$\begin{cases} a_t = \operatorname{argmax}_a (Q(s_t, a) + u(s_t, a)), \\ u(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N_r(s, b)}}{1 + N_r(s, a)} \end{cases} \quad (12.10)$$

其中, c_{puct} 是一个决定探索水平的常数。这种搜索控制策略开始偏爱高先验概率和低访问计数的走步, 然后逐渐偏爱高动作值的走步。

2. 扩展 (expansion)

当访问计数超过阈值且 $N_r(s, a) > n_{\text{thr}}$ 时, 后续状态 $s' = f(s, a)$ 被加到搜索树中当作新节点, 并初始化 $N(s', a) = N_r(s', a) = 0$, $W(s', a) = W_r(s', a) = 0$, $P(s', a) = P_\sigma(a, s')$ 。同时, 棋局状态 s' 被策略网络插入一个异步 GPU 评估队列中, 把软最大温度设置为 β , 用有监督策略网络更新先验概率, 即

$$P(s', a) \leftarrow P_\sigma^\beta(a | s') \quad (12.11)$$

另外, 还需要动态调整阈值 n_{thr} , 以保证棋局状态到策略队列的加入率与策略网络的 GPU 评估率相匹配。为了最小化端到端的评估时间, 棋局状态采用迷你块大小为 1 的策略网络和估值网络一起评价。

3. 评价 (evaluation)

对没有评价过的叶子棋局状态 s_L , 用估值网络计算其评价结果 $v_\theta(s_L)$ 。在每次模拟中, 第二个展开阶段都从叶节点 s_L 开始, 一直持续到对弈结束。对每个 $t \geq T$ 的步骤, 下棋双方都根据展开策略选择走步 $a_t \sim p_\pi(\cdot | s_t)$ 。当对弈达到结束状态时, 根据最后得分计算结果 $z_t = \pm r(s_T)$ 。



4. 回溯 (backup)

对模拟的每个树内步骤 $t \leq L$, 按输掉 n_{ul} 次对弈更新展开统计数据, $N_r(s_t, a_t) \leftarrow N_r(s_t, a_t) + n_{ul}$, $W_r(s_t, a_t) \leftarrow W_r(s_t, a_t) - n_{ul}$ 。这种虚拟输棋的方法可以压制其他线程同时探索相同的变化情况。在模拟结束时, 展开统计数据在回溯过程中通过每个 $t \leq L$ 的步骤进行更新, 把虚拟损失替换为新的结果 $N_r(s_t, a_t) \leftarrow N_r(s_t, a_t) - n_{ul} + 1$, $W_r(s_t, a_t) \leftarrow W_r(s_t, a_t) + n_{ul} + z_t$ 。在评价完叶子棋局状态 s_L 时, 还异步启动一个独立的反向传递过程。在另一个反向传递过程, 估值网络的输出 $v_\theta(s_L)$ 也会通过每个 $t \leq L$ 的步骤对估值统计数据更新: $N_v(s_t, a_t) \leftarrow N_r(s_t, a_t) + 1$, $W_v(s_t, a_t) \leftarrow W_v(s_t, a_t) + v_\theta(s_L)$ 。每个状态走步的总体评价是一种蒙特卡罗估计的加权平均: $Q(s, a) = (1 - \lambda) \frac{W_v(s, a)}{N_v(s, a)} + \lambda \frac{W_r(s, a)}{N_r(s, a)}$ 。这种平均利用加权参数 λ 对估值网络和展开评估一起进行最大化。所有更新都是无锁执行的。

作为 AlphaGo 的核心, APV-MCTS 算法是一种异步多线程搜索技术, 实际执行时需要在很多 CPU 上进行模拟, 同时在很多 GPU 上并行计算策略网络和估值网络。AlphaGo 的最终版本运行 APV-MCTS 算法, 需要使用 40 个搜索线程、48 个 CPU 和 8 个 GPU。

另外, AlphaGo 还有一个分布式版本, 运行 APV-MCTS 算法时需要使用很多机器、40 个搜索线程、1202 个 CPU 和 176 个 GPU。这个 AlphaGo 的分布式架构由一台主机和许多从机构成, 其中主机做主搜索, 从机通过远程 CPU 执行异步展开或通过远程 GPU 执行异步策略和估值网络的评估。整棵搜索树保存在主机上, 主机只执行模拟的树内阶段。叶节点棋局会传到从机的 CPU 执行模拟的展开阶段, 也会传到从机的 GPU 计算网络特征并对策略和估值网络进行评估。策略网络的先验概率会返回给主机, 代替新扩展节点保存的先验概率。来自展开和估值网络输出的奖励也会分别返回给主机, 用于所产生的搜索路径回溯。

12.3 AlphaGo Zero 的新思想

AlphaGo Zero 的横空出世标志着人工智能开始迈向了一个新的高度——自学成才。与 AlphaGo Fan 和 AlphaGo Lee 相比, AlphaGo Zero 的不同之处表现在几个重要方面。第一, 它只需要通过自弈强化学习进行训练, 从随机走子开始, 无须使用人类数据的指导。第二, 它只使用棋盘上的黑白棋子作为输入特征。第三, 它使用单一的神经网络, 而不是分离的策略网络和估值网络。第四, 它在这个单一神经网络的基础上使用更简单的树搜索评估棋局和走步, 无须执行任何蒙特卡罗展开策略。实现这些重要特点的关键在于, AlphaGo Zero 采用了一种新的强化学习算法, 在训练循环的内部集成了前向搜索, 从而可以迅速提高学习的准确度和稳定性。这种新方法使用一个以 θ 为参数的深层神经网络 f_θ , 输入棋局的原始表示 s 及其历史, 输出是各点的走步概率和一个评估值, 即 $(p, v) = f_\theta(s)$ 。走步概率向量表示选择一个走步 a 的概率 (a 可能是“过 (pass)”), 即 $p_a = \Pr(a | s)$ 。评估值 v 是一个标量, 表



示当前棋手赢棋的概率。网络 f_θ 的输入是一个 $19 \times 19 \times 17$ 的图像堆叠，由 17 个 19×19 的二值特征面组成，其中 8 个是当前棋手的相继历史棋子分布，8 个是当前对手的相继历史棋子分布，另一个是颜色特征面。虽然网络 f_θ 只具有单一结构，但是集成了策略网络和估值网络的功能，包含许多使用块归一化和校正非线性单元的卷积残差模块。

图 12.4 给出了 AlphaGo Zero 进行自弈强化学习的流水线说明，注意该图只是 19×19 棋盘的一小部分。网络 f_θ 是利用一种新型自弈强化学习算法来训练的，在每一个棋局 s 中都起着指导蒙特卡罗树搜索的作用。蒙特卡罗树搜索的输出是从当前棋局走每步的概率 π 。 π 是一个 361 维的向量，其分量值是通过执行很多次模拟搜索估算出来的，比如 1600 次模拟。与网络 f_θ 直接计算的走步概率 p 相比，搜索概率 π 通常有助于选择更好的走法。搜索自弈是一种强有力的评估手段，采用改进的蒙特卡罗树搜索策略选择走步，并把赢家 z 作为一个评估值样本。

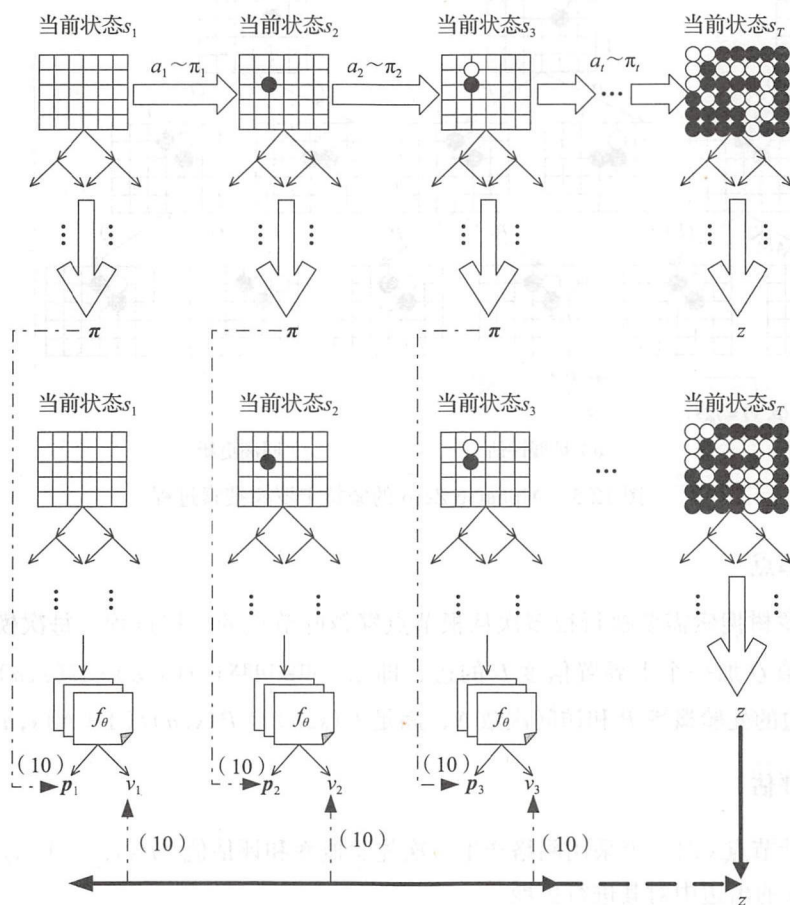


图 12.4 AlphaGo Zero 的网络自弈训练过程。其中， $s_1, s_2, \dots, s_t, \dots, s_T$ 表示各次自弈的棋局状态，输入以 θ 为参数的网络 f_θ 中，输出走步概率和评估值 $(p_i, v_i) = f_\theta(s_i)$ 。训练目标是使 p_i 尽量接近 π_i ， v_i 尽量接近最后的赢家 $z = \pm 1$



在自弈强化学习训练开始时,神经网络的参数被初始化为 θ_0 ,然后根据当前的棋局状态 s_t 利用 $f_{\theta_{t-1}}(s_t)$ 和蒙特卡罗树搜索估算 $\pi_t = \alpha_{\theta_{t-1}}(s_t)$ 进行自弈。如果自弈在步骤 T 结束,就存储最终的奖励得分 $r_T \in \{-1, +1\}$ 。中间步骤 t 存储的数据是 (s_t, π_t, z_t) ,其中 $z_t = \pm r_T$ 表示从当前棋手角度看到的最后赢家。这些自弈数据在所有步骤被均匀采样后用来训练网络的新参数 θ_i 。自弈强化学习的核心就是通过不断的循环迭代更新参数使走步概率和评估值 $(p, v) = f_{\theta_i}(s)$ 越来越接近搜索概率和自弈赢家 (π, z) ,逐步最小化下面的目标函数:

$$l = (z - v)^2 - \pi^T \log p + c \|\theta\|^2 \quad (12.12)$$

AlphaGo Zero 的蒙特卡罗树搜索过程如图 12.5 所示,与 AlphaGo 类似,也包含下面四个基本阶段。

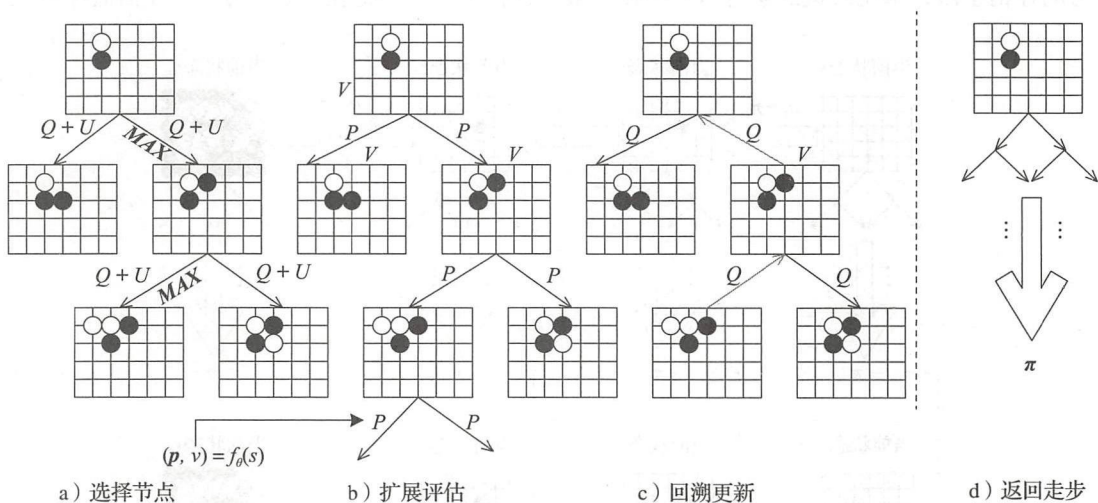


图 12.5 AlphaGo Zero 的蒙特卡罗树搜索过程

1. 选择节点

蒙特卡罗树搜索需要执行很多次从根节点穿越叶节点的模拟过程。每次模拟都选择具有最大动作值 Q 加一个上界置信度 U 的边,即 $a_t = \arg \max_a (Q(s, a) + U(s, a))$ 。其中, U 依赖于这条边的先验概率 P 和访问次数 N ,满足 $U(s, a) \propto P(s, a) / (1 + N(s, a))$ 。

2. 扩展评估

当达到叶节点 s 时,就采用网络产生一次先验概率和评估值 $(P(s, \cdot), V(s)) = f_{\theta}(s)$,并把 P 存储在 s 的出边中对其进行扩展。

3. 回溯更新

模拟过程穿越的每条边 (s, a) 都被更新以增加其访问次数 $N(s, a)$,而这些模拟的平均



估值则被用来重新计算动作值 $Q(s, a) = \frac{1}{N(s, a)} \sum_{s' | s, a \rightarrow s'} V(s')$ 。注意, $s, a \rightarrow s'$ 是指一次模拟在棋局状态 s 处采取动作 a 后最终到达 s' 。

4. 返回走步

蒙特卡罗树搜索计算走步概率向量 $\pi = \alpha_\theta(s)$, 其中每个分量 $\pi_a \propto N(s, a)^{1/\tau}$, 即正比于指数化的走步访问次数, 其中 τ 为温度参数。根据这个概率向量 π , 自弈过程走下一步棋。

12.4 仿效 AlphaGo 的围棋程序案例 MuGo

MuGo 是一款仿效 AlphaGo 的开源围棋程序, 用 Python 语言实现, 由 Brian Lee(brilee) 在 GitHub 上发布, 下载地址为 <https://github.com/brilee/MuGo>, 下载界面如图 12.6 所示。

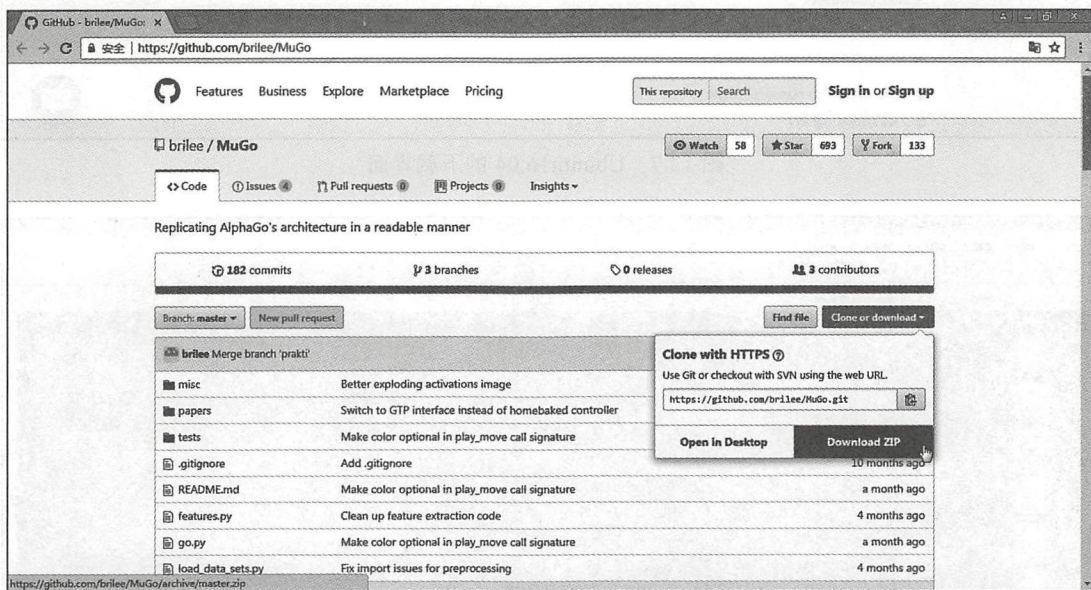


图 12.6 MuGo 的下载界面

下面分别介绍 MuGo 的开发环境、代码实现及说明、学习训练过程和运行演示效果。

12.4.1 MuGo 的开发环境

MuGo 可以运行在 Windows 或 Linux 系统平台上。这里选用 Linux 平台, 版本是 Ubuntu-16.04, 下载地址为 <https://www.ubuntu.com/download/desktop>, 下载界面如图 12.7 所示。

注意, 在安装 MuGo 之前, 还要安装 TensorFlow 和 GPU 驱动程序。如果实在没有 GPU 卡, 那就只能用 VMware 虚拟机暂时替代一下, 但运行速度等方面会受到影响。在虚拟机上



允许一台真实的计算机同时运行数个操作系统，Ubuntu 安装后的系统界面图 12.8 所示。

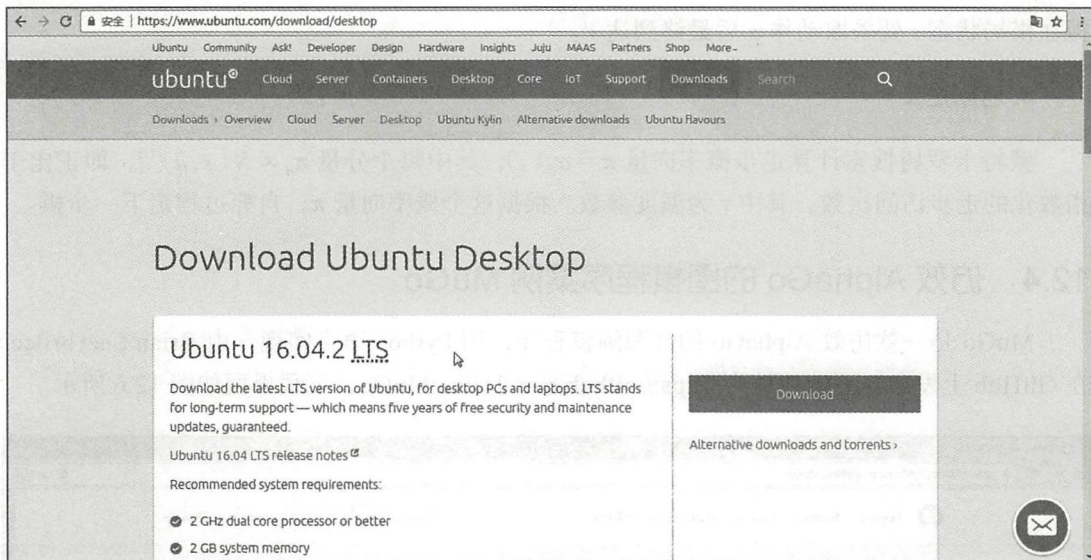


图 12.7 Ubuntu16.04 的下载界面

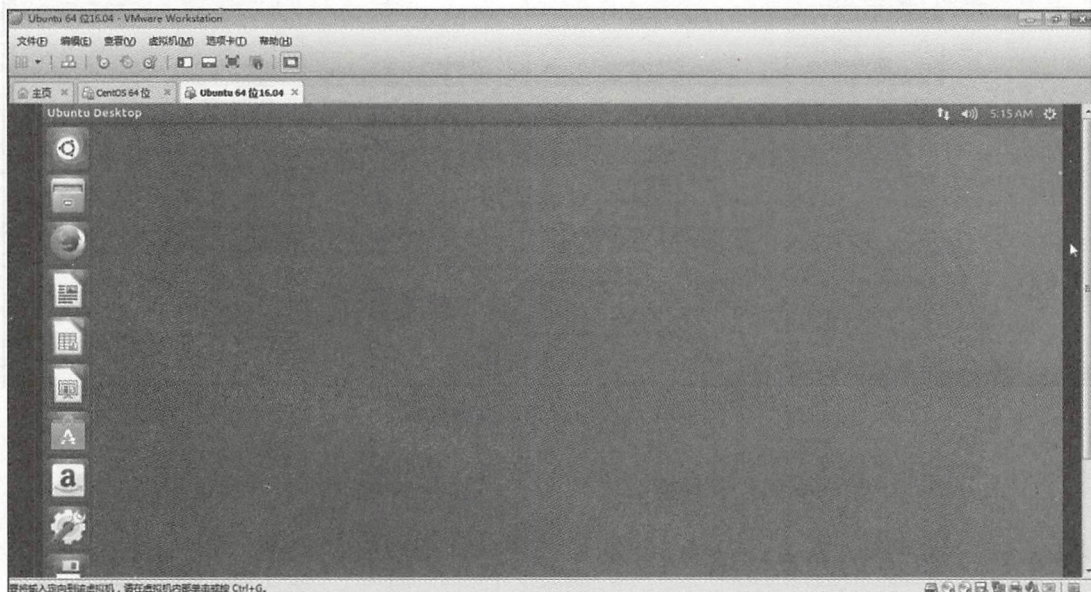


图 12.8 安装在虚拟机上的 Ubuntu 系统界面

与第 11 章讨论的笨笨鸟网络类似，围棋程序 MuGo 也是用 Python 语言开发的，但依赖的工具包有所不同。MuGo 依赖的是 Python 3.5.2、TensorFlow 1.1.0、PyGTK、argh、SFG 和 OpenJDK。其中，Python 3.5.2 是 Ubuntu 自带的，在 Ubuntu 系统中打开终端，输



入查询命令就可以查询 Python 的版本, 如图 12.9 所示。TensorFlow 的安装命令如图 12.10 所示。PyGTK 是一个创建 Python 图形用户界面的工具库, 包括 3 个安装包 pygtk、argh 和 SFG, 安装命令分别如图 12.11 ~ 图 12.13 所示。OpenJDK 是 Sun Microsystems 公司为 Java 平台构建的 Java 开发环境 (JDK) 的开源版本, 安装命令如图 12.14 所示。

```
but@ubuntu:~$ python --version
Python 2.7.12
but@ubuntu:~$ python3 --version
Python 3.5.2
but@ubuntu:~$
```

图 12.9 Python 的版本查询命令

```
but@ubuntu:~$ sudo pip3 install tensorflow
The directory '/home/but/.cache/pip/http' or its parent directory is not owned by
the current user and the cache has been disabled. Please check the permissions
and owner of that directory. If executing pip with sudo, you may want sudo's -H
flag.
The directory '/home/but/.cache/pip' or its parent directory is not owned by the
current user and caching wheels has been disabled. Check the permissions and ow
ner of that directory. If executing pip with sudo, you may want sudo's -H flag.
Collecting tensorflow
  Downloading tensorflow-1.1.0-cp35-cp35m-manylinux1_x86_64.whl (31.0MB)
    11% |#####| 3.4MB 554KB/s eta 0:00:50
```

图 12.10 TensorFlow 的安装

```
but@ubuntu:~$ sudo pip3 install pygtk
[sudo] password for but:
The directory '/home/but/.cache/pip/http' or its parent directory is not owned b
y the current user and the cache has been disabled. Please check the permissions
and owner of that directory. If executing pip with sudo, you may want sudo's -H
flag.
The directory '/home/but/.cache/pip' or its parent directory is not owned by the
current user and caching wheels has been disabled. Check the permissions and ow
ner of that directory. If executing pip with sudo, you may want sudo's -H flag.
Collecting pygtk
  Downloading pygtk-0.4.tar.gz
Installing collected packages: pygtk
  Running setup.py install for pygtk ... done
Successfully installed pygtk-0.4
but@ubuntu:~$
```

图 12.11 pygtk 的安装

```
but@ubuntu:~$ sudo pip3 install argh
The directory '/home/but/.cache/pip/http' or its parent directory is not owned b
y the current user and the cache has been disabled. Please check the permissions
and owner of that directory. If executing pip with sudo, you may want sudo's -H
flag.
The directory '/home/but/.cache/pip' or its parent directory is not owned by the
current user and caching wheels has been disabled. Check the permissions and ow
ner of that directory. If executing pip with sudo, you may want sudo's -H flag.
Collecting argh
  Downloading argh-0.26.2-py2.py3-none-any.whl
Installing collected packages: argh
Successfully installed argh-0.26.2
but@ubuntu:~$
```

图 12.12 argh 的安装

```
but@ubuntu:~$ sudo pip3 install -v sgf==0.5
[sudo] password for but:
The directory '/home/but/.cache/pip/http' or its parent directory is not owned b
y the current user and the cache has been disabled. Please check the permissions
and owner of that directory. If executing pip with sudo, you may want sudo's -H
flag.
The directory '/home/but/.cache/pip' or its parent directory is not owned by the
current user and caching wheels has been disabled. Check the permissions and ow
ner of that directory. If executing pip with sudo, you may want sudo's -H flag.
Collecting sgf==0.5
  1 location(s) to search for version(s) of sgf:
    * https://pypi.python.org/simple/sgf/
    Getting page https://pypi.python.org/simple/sgf/
    Looking up "https://pypi.python.org/simple/sgf/" in the cache
    No cache entry available
```

图 12.13 SGF 的安装

```
but@ubuntu:~/MuGo-master/gogui-1.4.9$ sudo apt-get install openjdk-8-jdk
[sudo] password for but:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
ca-certificates-java fonts-dejavu-extra java-common libgif7 libice-dev
```

图 12.14 OpenJDK 的安装

12.4.2 MuGo 的代码实现及说明

MuGo 是一个用 Python 实现 AlphaGo 模型的 Github 开源项目，主要实现了策略网络、快速走步策略网络，以及蒙特卡罗树搜索。其中，策略网络和快速走步网络的结构是完全一样的。但需要指出的是，MuGo 只是对 AlphaGo 的一个复现，并没有实现 AlphaGo 的所有模块，特别是并没有实现估值网络。另外，MuGo 虽然实现了蒙特卡罗树及其搜索过程，但是因速度太慢在评估棋局状态时并未使用。MuGo 采用多次直到中局的展开对叶节点直接进行评估。

把下载的 MuGo 解压后，主目录为 MuGo-master，共有 7 个文件，其中 5 个关键文件分别是 go.py、features.py、policy.py、strategies.py 和 selfplay.py，它们的功能描述如表 12.1 所示。MuGo 的技术核心是通过这 5 个关键文件实现的，下面对它们一一介绍。

表 12.1 MuGo 程序的关键文件及其功能

关键文件	功 能
go.py	用来表示整个棋盘
features.py	用来提取特征
policy.py	实现策略网络和快速走步网络
strategies.py	实现蒙特卡罗树及其搜索过程，并且使用快速走步策略对叶节点进行展开（模拟），以及对叶节点的评估
selfplay.py	实现自弈强化学习网络

1. go.py 文件的代码实现及说明

go.py 文件代表 19 路棋盘，具有初始化棋盘、显示棋子分布状态、判别走步合法性等基本功能。这个文件主要包含 3 个类：Group 类、Position 类和 LibertyTracker 类。其中，Group 类用来表示博弈双方的棋子分组，每方的棋子都是按组为单位进行管理的，每个组包括 4 个部分——组号、组子、组气和组色。Position 类用来表示棋盘的当前状态。LibertyTracker 是一个与（棋子）组相关的“气”类，用来判断组的死活。每个组都是靠“气”来生存的，没有“气”就意味着被对方“围死”。每个组的气是动态变化的，己方着子和对方着子都有可能改变双方某个组或者某些组的“气”。下面分别介绍 Group 类、Position 类和 LibertyTracker 类的代码实现及说明。

（1）Group 类

```
Class Group(namedtuple('Group', ['id', 'stones', 'liberties', 'color'])):    # 组类，
# 包括组号、组子、组气和组色
```

```

    Def __eq__(self, other):    # 如果所创建的组合法, 则将其各属性值都初始化为 other 的相应
    # 属性值
        return self.stones == other.stones and self.liberties == other.\
liberties and self.color == other.color

```

注意: 在同一个组中, 每颗棋子具有相同的组号 “id”、相同的组气 “liberties” 和相同的组色 “color”。

(2) Position 类

```

class Position():    # 棋盘状态类
    def __init__(self, board=None, n=0, komi=7.5, caps=(0,0), lib_tracker=None,
ko=None, recent=tuple(), to_play=BLACK):
        self.board = board if board is not None else np.copy(EMPTY_BOARD)
        # board 表示棋盘
        self.n = n                # 表示对战回合数
        self.komi=komi            # 表示“贴目”, 按照中国围棋规则, 黑棋要贴 7.5 目给白棋
        self.caps=caps            # 记录下棋过程的双方提子情况
        self.lib_tracker=lib_tracker or LibertyTracker.from_board(self.board)
        # 表示棋子的“气”
        self.ko=ko                # 表示“劫”, “打劫”可能会陷入无限死循环, 会受到一定限制
        self.recent=recent        # 表示近期着子记录, 包括两项: 下棋选手(黑/白)、着子位置
        self.to_play=to_play      # 着子选手, 要么着黑子, 要么着白子

    def __deepcopy__(self, memodict={}):    # 对象的深拷贝, 相当于新创建一个本类对象
        new_board = np.copy(self.board)    # 拷贝棋盘
        new_lib_tracker = copy.deepcopy(self.lib_tracker) # 拷贝棋盘上棋子的“气”
        return Position(new_board, self.n, self.komi, self.caps, new_lib_\
tracker, self.ko, self.recent, self.to_play)

    def is_move_suicidal(self, move):        # 判断着子是否为“自杀”, “自杀”是不允许的
        potential_libs=set()                # 记录棋子的“气”
        for n in NEIGHBORS[move]:           # 遍历着子位置的上下左右 4 个直接相邻位置
            neighbor_group_id=self.lib_tracker.group_index[n] # 找到着子位置的组号
            if neighbor_group_id==MISSING_GROUP_ID:           # 如果组号为空
                return False # 着子位置不是“自杀”, 可以着子
            neighbor_group=self.lib_tracker.groups[neighbor_group_id]
            # 根据组号获得组信息
            if neighbor_group.color==self.to_play:            # 如果组色与当前棋手一致
                potential_libs|=neighbor_group.liberties      # 求并集, 收集组气
            elif len(neighbor_group.liberties) == 1:           # 否则, 如果组气为 1
                return False # 在当前位置着子将会提掉对方的一个组
        potential_libs-=set([move])            # 做集合减法, 除去 move
        return not potential_libs              # 返回是否有“气”

    def is_move_legal(self, move):            # 判断“着子”是否合法
        if move is None:                     # 不走合法, 相当于“过”
            return True
        if self.board[move]!= EMPTY:         # 在有子的位置着子不合法
            return False
        if move == self.ko:                   # 着子后“打劫”不合法
            return False
        if self.is_move_suicidal(move):      # “自杀”着子不合法

```



```

        return False
    return True                # 其他走子都合法

def pass_move(self, mutate=False):    # 轮到己方着子的时候，选择“过”，让对方着子
    pos=self if mutate else copy.deepcopy(self)
    # 默认把 Position() 深拷贝一份，是个全新的对象
    pos.n+=1                    # 对战回合数自加
    pos.recent+=(PlayerMove(pos.to_play, None),)    # 记录着子位置为空
    pos.to_play*=-1            # 更换走子方
    pos.ko=None                # 无“打劫”
    return pos                 # 返回深拷贝对象

def flip_playerturn(self, mutate=False):
    pos = self if mutate else copy.deepcopy(self)
    # 默认把 Position() 深拷贝一份，是个全新的对象
    pos.ko=None                # 无“打劫”
    pos.to_play*=-1            # 更换走子方
    return pos                 # 返回深拷贝对象

def get_liberties(self):    # 返回已经着子位置的“气”，即组气
    return self.lib_tracker.liberty_cache

def play_move(self, c, color=None, mutate=False):    # 遵守中国围棋规则走子
    if color is None:    # 如果 color 值为空，则“默认”玩家下
        color=self.to_play    # 设置着子的玩家
    pos=self if mutate else copy.deepcopy(self)
    # 默认把 Position() 深拷贝一份
    if c is None:    # 如果走子位置为空，就视为“过”
        pos=pos.pass_move(mutate=mutate)
        return pos    # 返回深拷贝对象
    if not self.is_move_legal(c):    # 如果走步位置不合法
        raise IllegalMove("Move at {} is illegal: \n{}".format(c, self))
    # 终止程序
    place_stones(pos.board, color, [c])
    # 在位置 c 处走步，并修改 c 处的 color 标志值
    captured_stones=pos.lib_tracker.add_stone(color, c)    # 对方可能被提子的位置集合
    place_stones(pos.board, EMPTY, captured_stones)
    # 重新设置这些位置的 color 标志为空
    opp_color=color*-1    # 更换走子方
    if len(captured_stones)==1 and is_koish(self.board, c)==opp_color:
        new_ko = list(captured_stones)[0]
        # 只提掉对方一子且 c 处周围都是对方棋子，形成“劫”
    else:
        new_ko = None    # 否则无“劫”
    if pos.to_play==BLACK:    # 如果是“黑方”走子
        new_caps=(pos.caps[0]+len(captured_stones), pos.caps[1])
        # 记录被提掉的白子个数
    else:
        new_caps=(pos.caps[0], pos.caps[1]+len(captured_stones))
        # 记录被提掉的黑子个数
    pos.n += 1    # 对战回合数自加
    pos.caps=new_caps    # 记录被对方“吃掉”的棋子个数
    pos.ko=new_ko    # 记录是否存在“劫”

```

```

pos.recent+=(PlayerMove(color,c),) # 记录当前玩家的近期走子情况
pos.to_play*=-1 # 更换走子方
return pos # 返回当前状态

def score(self): # 黑子个数及围死的空白个数、白子个数及围死的空白个数、贴目数
    working_board=np.copy(self.board) # 将棋盘对象拷贝一份
    while EMPTY in working_board: # 找到棋盘上的空白部分
        unassigned_spaces=np.where(working_board==EMPTY) # 找到棋盘上所有空白位置
        c=unassigned_spaces[0][0],unassigned_spaces[1][0]
        # 找出第一个满足条件的空白位置
    territory,borders=find_reached(working_board,c)
    # territory 表示空白, borders 表示非空白
    border_colors=set(working_board[b] for b in borders) # 得到空白边界的着子颜色值
    X_border=BLACK in border_colors # 判断空白边界部分是否有黑棋
    O_border=WHITE in border_colors # 判断空白边界部分是否有白棋
    if X_border and not O_border: # 如果空白边界只有黑棋、没有白棋
        territory_color=BLACK # 就把空白区域归为黑棋领地
    elif O_border and not X_border: # 否则,如果空白边界只有白棋、没有黑棋
        territory_color=WHITE # 就把空白区域归为白棋领地
    else: # 再则,如果边界是黑白混合
        territory_color=UNKNOWN # 就不把空白区域归属任何一方
    place_stones(working_board,territory_color,territory)
    # 标记空白区域的归属
    return np.count_nonzero(working_board==BLACK)-np.count_nonzero(working\
_board==WHITE) -self.komi # 返回黑棋的最终得分

def result(self): # 计算黑棋玩家的分数
    score=self.score()
    if score>0:
        return 'B+' + '%.1f' % score # 黑棋吃掉白棋,黑棋得正分
    elif score<0:
        return 'W+' + '%.1f' % abs(score) # 白棋吃掉黑棋,黑棋得负分
    else:
        return 'DRAW' # 不吃棋的情况

set_board_size(19) # 初始化棋盘为 19 路

```

(3) LibertyTracker 类

```

class LibertyTracker(): # “气”类,用来判断(棋子)组的死活
    @staticmethod # 这种标记方法可以通过类名来进行引用,而不需要通过类对象
    def from_board(board): # 传入一个 19×19 的棋盘数组,取值范围为 -1~4
        board=np.copy(board) # 拷贝一份棋盘数组,只是用来模拟,实际并不改变原棋局
        curr_group_id =0 # 初始化当前组的编号 id 为 0
        lib_tracker=LibertyTracker() # 构造默认“气”类
        for color in (WHITE,BLACK): # 对白子和黑子进行分组
            while color in board: # 如果棋盘上有黑子或者白子
                curr_group_id+=1 # 组编号 id 自加
                found_color=np.where(board==color)
                # 在整个棋盘上寻找到所有值为 color 的位置
                coord=found_color[0][0],found_color[1][0]
                # 得到第一个颜色值为 color 的位置。
                chain,reached=find_reached(board,coord)

```

```

# 用 chain 保存组，用 reached 保存组边界
liberties=set(r for r in reached if board[r]==EMPTY)
# 找出组边界的空白位置
new_group=Group(curr_group_id,chain,liberties,color)
# 创建一个新组
lib_tracker.groups[curr_group_id]=new_group
# 将新组加入棋盘分组中
for s in chain:
    lib_tracker.group_index[s]=curr_group_id
    # 同组棋子都标记相同的组 id
    place_stones(board,FILL,chain) # 把有子的位置标记为 FILL
lib_tracker.max_group_id=curr_group_id # 记录最大组号
liberty_counts=np.zeros([N, N],dtype=np.uint8)
# 创建一个棋盘大小的全 0 矩阵，存放“气”
for group in lib_tracker.groups.values(): # 遍历棋盘上的所有组
    num_libs=len(group.liberties) # 计算当前组的“气”
    for s in group.stones: # 遍历组内所有棋子
        liberty_counts[s]=num_libs # 设置组内所有棋子为相同的“气”
lib_tracker.liberty_cache=liberty_counts # 更新棋盘上所有棋子的“气”
return lib_tracker # 返回当前棋盘状态

def __init__(self, group_index=None, groups=None, liberty_cache=None,
max_group_id=1): # 构造函数
    # group_index 表示棋盘上的组号，默认值 -1 表示还没有添加到组
    # groups 表示棋盘上所有棋子的分组，或有子位置的分组
    # liberty_cache 记录棋盘上所有棋子的“气”，无子位置的“气”默认为 0
    # max_group_id 用来记录棋盘的分组数目
    self.group_index = group_index if group_index is not None else -np.\
ones([N, N], dtype=np.int32)
    self.groups = groups or {}
    self.liberty_cache = liberty_cache if liberty_cache is not None else\
np.zeros([N, N], dtype=np.uint8)
    self.max_group_id = max_group_id
def __deepcopy__(self, memodict={}): # 本类对象的深拷贝，用来创建一个新对象
    new_group_index=np.copy(self.group_index)
    new_lib_cache = np.copy(self.liberty_cache)
    new_groups = {
        group.id: Group(group.id, set(group.stones), set(group.liberties),
        group.color)
        for group in self.groups.values()
    }
    return LibertyTracker(new_group_index, new_groups, liberty_cache=new\
_lib_cache, max_group_id=self.max_group_id)

def add_stone(self,color,c): # 从位置 c 开始对棋子分组
    assert self.group_index[c]==MISSING_GROUP_ID
    # 找到位置 c 的组号，否则异常结束
    captured_stones=set() # 记录可能被“吃掉”的对方棋子集合
    opponent_neighboring_group_ids=set() # 保存“对手”的所有组
    friendly_neighboring_group_ids=set() # 保存“己方”的所有组
    empty_neighbors=set() # 保存所有空白位置
    for n in NEIGHBORS[c]: # 遍历 c 处上下左右的 4 个直接邻接位置
        neighbor_group_id=self.group_index[n] # 找到这些位置的组号

```



```

if neighbor_group_id!=MISSING_GROUP_ID:      # 如果找到组号
    neighbor_group=self.groups[neighbor_group_id] # 根据组号获取组信息
    if neighbor_group.color==color:
        # 如果c处棋子的颜色与直接相邻组的颜色相同
        friendly_neighboring_group_ids.add(neighbor_group_id)
        # 该组为“己方”组
    else:
        opponent_neighboring_group_ids.add(neighbor_group_id)
        # 否则为“对手”组
else:
    empty_neighbors.add(n) # 保存空白位置n
new_group=self._create_group(color,c,empty_neighbors)
# 创建一个只在c处有棋子的新组
for group_id in friendly_neighboring_group_ids: # 遍历找到所有“己方”组
    new_group=self._merge_groups(group_id,new_group.id)
    # 合并满足相邻条件的组
for group_id in opponent_neighboring_group_ids: # 遍历找到所有“对手”组
    neighbor_group=self.groups[group_id] # 根据对手组号获取组信息
    if len(neighbor_group.liberties)==1:
        # 如果在c处走子前对手组的“气”为1
        captured=self._capture_group(group_id)
        # 己方走子后导致对手组的“气”为0，被提子
        captured_stones.update(captured) # 把被提子保存为一个新集合
    else:
        self._update_liberties(group_id,remove={c}) # 否则更新对手组的“气”
        self._handle_captures(captured_stones) # 保存对方所有被“吃掉”的棋子
        if len(new_group.liberties)==0: # 如果合并组的气为0，表示自杀，不合法
            raise IllegalMove("Move at {} would commit suicide!\n".format(c))
        # 提示自杀
return captured_stones # 否则返回对方所有被吃掉的棋子位置

def _create_group(self,color,c,liberties): # 创建一个新的组
    self.max_group_id += 1 # 最大组号自增
    new_group=Group(self.max_group_id,set([c]),liberties,color)
    # 创建只包含一个棋子的新组
    self.groups[new_group.id]=new_group # 将新组加入棋盘分组中
    self.group_index[c]=new_group.id # 设置c处棋子的组号
    self.liberty_cache[c]=len(liberties) # 计算c处棋子的“气”
    return new_group # 返回新组

def _merge_groups(self,group1_id,group2_id): # 合并己方相邻组
    group1=self.groups[group1_id] # 得到第1组的信息
    group2=self.groups[group2_id] # 得到第2组的信息
    group1.stones.update(group2.stones)
    # 将第2组的棋子先合并到第1组的棋子集合中
    del self.groups[group2_id] # 从棋盘分组中删除第2组
    for s in group2.stones:
        self.group_index[s]=group1_id
    # 将棋盘分组中原第2组的组号更新为第1组
    self._update_liberties(group1_id, add=group2.liberties,
        remove=(group2.stones | group1.stones))
    return group1 # 返回合并组

def _capture_group(self,group_id): # 按组提掉对方棋子
    dead_group=self.groups[group_id] # 获取将要被“提掉”的对方组

```

```

del self.groups[group_id] # 从棋盘分组中删除被提组
for s in dead_group.stones: # 更新被提组的位置属性
    self.group_index[s]=MISSING_GROUP_ID # 将位置的组号设置为-1,表示无组号
    self.liberty_cache[s]=0 # 将位置的“气”设置为0
return dead_group.stones # 返回被提组
def _update_liberties(self,group_id,add=None,remove=None): # 更新某个组的气
    group=self.groups[group_id] # 根据组号获取组信息
    if add: # 如果是合并组
        group.liberties.update(add) # 更新合并组的气
    if remove:
        group.liberties.difference_update(remove) # 去掉重复计算的气
    new_lib_count=len(group.liberties) # 计算该组的气
    for s in group.stones:
        self.liberty_cache[s]=new_lib_count # 更新组内所有棋子的气
def _handle_captures(self,captured_stones): # 处理被提对手组的棋子
    for s in captured_stones: # 遍历对手被提棋子的位置
        for n in NEIGHBORS[s]: # 遍历被提子上下左右的直接相邻位置
            group_id=self.group_index[n] # 得到这些位置的组号
            if group_id!= MISSING_GROUP_ID: # 有组号
                self._update_liberties(group_id,add={s}) # 更新这个组的“气”

```

2. features.py文件的代码实现及说明

features.py 文件主要用于提取棋盘状态的 28 个特征（如表 12.2 所示），但 AlphaGo 提取了 48 个特征。

下面是对 features.py 的详细描述。

表 12.2 MuGo 案例使用的 28 个特征

特征	特征平面数 (plane)	描述
stone_color_feature	3	分别表示当前棋局中的黑棋、白棋和空白
ones_feature	1	全 1 常量特征
recent_move_feature	8	表示当前黑白双方的棋子分布
liberty_feature	8	表示当前棋局棋子“气”的特征
would_capture_feature	8	表示当前棋局可能被吃掉的棋子特征

```

import numpy as np
import go
from utils import product
P=8 # “热编码”的长度为 8
def make_onehot(feature, planes): # 创建热编码
    onehot_features=np.zeros(feature.shape + (planes,), dtype=np.uint8)
    # 创建 19×19×8 的特征平面
    capped=np.minimum(feature,planes) # 计算 feature 和 planes 的对应元素最小值
    onehot_index_offsets=np.arange(0,product(onehot_features.\
    shape),planes)+capped.ravel() # 热编码偏移量
    nonzero_elements=(capped!= 0).ravel() # 即将被吃掉的棋子
    nonzero_index_offsets=onehot_index_offsets[nonzero_elements]-1

```

```

# 即将被吃棋子的热编码偏移量
onehot_features.ravel()[nonzero_index_offsets]=1 # 进行热编码
return onehot_features # 返回8个热编码的特征面

def planes(num_planes): # 传入特征面的个数
    def deco(f): # f 可以看成隐参数
        f.planes = num_planes
        return f
    return deco

@planes(3)
def stone_color_feature(position):
    board=position.board # 获取棋盘
    features=np.zeros([go.N,go.N,3],dtype=np.uint8)
    # 创建3个特征面表示棋盘的黑棋、白棋和空白
    if position.to_play==go.BLACK: # 如果当前走子方是黑棋选手
        features[board==go.BLACK,0]=1 # 第一个特征面下过黑子的位置置1
        features[board==go.WHITE,1]=1 # 第二个特征面下过白子的位置置1
    else: # 反之,当前走子方是白棋选手
        features[board == go.WHITE, 0] = 1 # 第一个特征面下过白子的位置置1
        features[board == go.BLACK, 1] = 1 # 第二个特征面下过黑子的位置置1
    features[board==go.EMPTY,2]=1 # 第三个特征面在空白位置置1
    return features

@planes(1)
def ones_feature(position): # 创建全1特征面
    return np.ones([go.N, go.N, 1], dtype=np.uint8) # 创建大小为19×19×1的全1特征面

@planes(P)
def recent_move_feature(position):
    onehot_features = np.zeros([go.N,go.N,P], dtype=np.uint8)
    # 初始化8个全0特征面
    for i, player_move in enumerate(reversed(position.recent[-P:])):
        # 连续获取8次玩家走子记录
        _, move=player_move # 获取走子位置
        if move is not None: # 如果位置非空(空即“过”)
            onehot_features[move[0],move[1],i]=1 # 在第i个特征面的当前位置置1
    return onehot_features

@planes(P)
def liberty_feature(position): # 棋子“气”的特征
    return make_onehot(position.get_liberties(), P)
    # 用8个19×19的特征面表示棋子“气”的特征

@planes(P)
def would_capture_feature(position): # 当前棋局可能被吃掉的棋子特征
    features=np.zeros([go.N,go.N],dtype=np.uint8) # 初始化19×19的全0数组
    for g in position.lib_tracker.groups.values(): # 遍历棋盘上的所有组
        if g.color==position.to_play: # 如果组色与当前棋手一致
            continue # 跳过
        if len(g.liberties)==1: # 如果组“气”等于1
            last_lib=list(g.liberties)[0] # 得到这口“气”的位置
            features[last_lib]+=len(g.stones) # 用feature记录当前位置组的大小

```



```

return make_onehot(features,P)                                # 对 feature 进行热编码

DEFAULT_FEATURES=[
    stone_color_feature,
    ones_feature,
    liberty_feature,
    recent_move_feature,
    would_capture_feature,
]    # 28 个特征的数据结构

def extract_features(position, features=DEFAULT_FEATURES):    # 提取棋局的特征
    return np.concatenate([feature(position) for feature in features],axis=2)
# 拼接 28 个默认特征

def bulk_extract_features(positions, features=DEFAULT_FEATURES):
# 提取多个棋局的特征
    num_positions=len(positions)                                # 计算棋局个数
    num_planes = sum(f.planes for f in features) # num_planes=28
    output=np.zeros([num_positions, go.N, go.N, num_planes],dtype=np.uint8)
    for i,pos in enumerate(positions):
        output[i]=extract_features(pos,features=features)
    return output

```

3. policy.py 文件的代码实现及说明

policy.py 描述的是一个策略网络，包括输入层、卷积隐含层、全连接隐含层、softmax 输出层。其中，输入层的结构为 $19 \times 19 \times 28$ ，也就是 28 个 19×19 的特征面。卷积隐含层依次为 1 个残差卷积结构（具有两个卷积核，大小分别为 5×5 和 1×1 ）、12 个卷积层（都具有 2 个卷积核，大小均为 3×3 ），以及另一个残差卷积结构。全连接隐含层含有 128 个神经元。卷积隐含层和全连接隐含层的激活函数都是 ReLU（校正线性单元）。输出层为 softmax 软最大分类输出。另外，policy.py 只包含一个类，即 PolicyNetwork 类。下面进行详细介绍。

PolicyNetwork 类

```

class PolicyNetwork(object):                                # 策略网络
    def __init__(self, k=128, num_int_conv_layers=11, use_cpu=False):
# 构造函数
        self.num_input_planes=sum(f.planes for f in features.DEFAULT_FEATURES)
                                                # 28 个特征面
        self.k=k                                            # 每一层的宽度
        self.num_int_conv_layers=num_int_conv_layers      # 设置卷积层的层数为 11
        self.test_summary_writer=None                    # 汇总测试参数，用于 tensorboard 可视化
        self.training_summary_writer=None                # 汇总训练参数，用于 tensorboard 可视化
        self.test_stats=StatisticsCollector()             # 测试时汇总损失和准确率
        self.training_stats = StatisticsCollector()        # 训练时汇总损失和准确率
        self.session=tf.Session()                        # 上下文会话对象，用于初始化网络参数
        if use_cpu:                                        # 表示使用 CPU 来完成计算任务
            with tf.device("/cpu:0"):                      # 选择 CPU
                self.set_up_network()

```

```

else:
    # 如果有 GPU, 自动选择 GPU 加速
    self.set_up_network()

def set_up_network(self):
    # 创建策略网络
    global_step=tf.Variable(0, name="global_step", trainable=False)
    # 监督训练次数
    RL_global_step = tf.Variable(0, name="RL_global_step", trainable=False)
    # 强化训练次数
    x=tf.placeholder(tf.float32, [None,go.N,go.N,self.num_input_planes])
    # 网络输入 (19×19×28)
    y=tf.placeholder(tf.float32,shape=[None,go.N ** 2])
    # 专家走子位置
    reinforce_direction = tf.placeholder(tf.float32, shape=[])
    # 强化方向, -1 表示减弱, 1 表示增强
    def _weight_variable(shape,name):
        # 初始化网络的权重
        number_inputs_added=utils.product(shape[:-1])
        stddev = 1/math.sqrt(number_inputs_added)
        # 计算标准差
        return tf.Variable(tf.truncated_normal(shape, stddev=stddev),
            name=name)
    def _conv2d(x, W):
        # 定义卷积运算, 步长为 1、SAME 表示输入和输出大小相同
        return tf.nn.conv2d(x, W, strides=[1,1,1,1], padding="SAME")
    W_conv_init55=_weight_variable([5,5,self.num_input_planes,self.k],name="W_conv_init55")
    W_conv_init11=_weight_variable([1,1,self.num_input_planes,self.k],name="W_conv_init11")
    h_conv_init=tf.nn.relu(_conv2d(x, W_conv_init55) +_conv2d(x, W_conv_init11), name="h_conv_init")
    W_conv_intermediate=[]
    h_conv_intermediate=[]
    # 定义残差模块的卷积核
    # 定义残差模块的输出
    # 构造第 1 个残差模块的输入
    _current_h_conv=h_conv_init
    for i in range(self.num_int_conv_layers):
        # 循环构造卷积层
        with tf.name_scope("layer"+str(i)):
            _resnet_weights1=_weight_variable([3,3,self.k,self.k],name="W_conv_resnet1")
            # 3×3 的核
            _resnet_weights2=_weight_variable([3,3,self.k,self.k],name="W_conv_resnet2")
            # 3×3 的核
            _int_conv=tf.nn.relu(_conv2d(_current_h_conv,_resnet_weights1), name="h_conv_intermediate")
            # 卷积之后 ReLU 激活
            _output_conv=tf.nn.relu(_current_h_conv+_conv2d(_int_conv,_resnet_weights2), name="h_conv")
            # 再次卷积之后 ReLU 激活
            W_conv_intermediate.extend([_resnet_weights1,_resnet_weights2])
            # 保存卷积核
            h_conv_intermediate.append(_output_conv)
            # 保存输出
            _current_h_conv=_output_conv
            # 把当前残差模块的输出作为下一模块的输入
    W_conv_final=_weight_variable([1,1,self.k, 1], name="W_conv_final")
    # 初始化全连接权重
    b_conv_final=tf.Variable(tf.constant(0, shape=[go.N ** 2], dtype=tf.float32),
        name="b_conv_final")
    h_conv_final=_conv2d(h_conv_intermediate[-1], W_conv_final)
    # 计算全连接层输出
    logits=tf.reshape(h_conv_final, [-1, go.N ** 2]) + b_conv_final
    # 改变全连接层输出的形态

```

```

log_likelihood_cost=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
learning_rate=tf.train.exponential_decay(1e-2, global_step,4*10**6,0.5)
# 学习率衰减方式
train_step=tf.train.GradientDescentOptimizer(learning_rate).\
minimize(log_likelihood_cost, global_step=global_step)
# 用梯度下降优化 SL 参数
was_correct = tf.equal(tf.argmax(logits,1),tf.argmax(y,1))
# 用网络输出与标签值进行匹配
accuracy=tf.reduce_mean(tf.cast(was_correct,tf.float32))
# 计算平均准确率
reinforce_step=tf.train.GradientDescentOptimizer(1e-2).minimize(log_likelihood_cost *reinforce_direction, global_step=RL_global_step)
# 用梯度下降优化 RL 参数
weight_summaries = tf.summary.merge([tf.summary.histogram(weight_var.\
name, weight_var) for weight_var in [W_conv_init55,W_conv_init11] + \
W_conv_intermediate+ [W_conv_final, b_conv_final]],name="weight_summaries")
# 保存网络训练的所有参数，用于可视化
activation_summaries = tf.summary.merge([tf.summary.histogram(act_var.\
name, act_var)
    for act_var in [h_conv_init] + h_conv_intermediate + [h_conv_final]],
    name="activation_summaries") # 保存网络所有层的输出，用于可视化
saver=tf.train.Saver() # 保存网络的权重、偏置等参数，或恢复网络参数
for name, thing in locals().items():
    if not name.startswith('_'):
        setattr(self, name, thing)

def initialize_logging(self, tensorboard_logdir): # 初始化日志文件，用于可视化
    self.test_summary_writer=tf.summary.FileWriter(os.path.\
join(tensorboard_logdir,"test"),self.session.graph)
    # 指明测试过程中的网络参数存放路径
    self.training_summary_writer=tf.summary.FileWriter(os.path.\
join(tensorboard_logdir,"training"),self.session.graph)
    # 指明训练过程中的网络参数存放路径
def initialize_variables(self, save_file=None): # 初始化网络参数的方法
    self.session.run(tf.global_variables_initializer()) # 初始化所有变量
    if save_file is not None:
        self.saver.restore(self.session, save_file) # 恢复保存在本地的网络参数
def get_global_step(self): # 获取网络参数的训练次数
    return self.session.run(self.global_step)
def save_variables(self, save_file): # 保存训练好的网络参数
    if save_file is not None:
        print("Saving checkpoint to %s" % save_file, file=sys.stderr)
        self.saver.save(self.session, save_file)

def train(self, training_data, batch_size=32):
    # 每次默认以 32 个样本的批大小进行训练
    num_minibatches=training_data.data_size//batch_size # 训练批的个数
    for i in range(num_minibatches): # 按批训练
        batch_x, batch_y = training_data.get_batch(batch_size)

```



```

        # 按批获取棋局状态和标签
        _, accuracy, cost = self.session.run([self.train_step, self.
        accuracy, self.log_likelihood_cost], feed_dict={self.x: batch_x,
        self.y: batch_y, self.reinforce_direction: 1}) # 执行训练过程
        self.training_stats.report(accuracy, cost)
        # 保存准确率和损失, 用于可视化
    avg_accuracy, avg_cost, accuracy_summaries = self.training_stats.collect()
    # 获取可视化指标
    global_step = self.get_global_step()
    # 获取训练次数
    print("Step %d training data accuracy: %g; cost: %g" % (global_
    step, avg_accuracy, avg_cost))
    if self.training_summary_writer is not None: # 保存每次训练网络的所有结果
        activation_summaries = self.session.run(self.activation_summaries,
        feed_dict={self.x: batch_x, self.y: batch_y, self.reinforce_
        direction: 1})
    self.training_summary_writer.add_summary(activation_summaries, global_step)
    self.training_summary_writer.add_summary(accuracy_summaries,
    global_step)

def reinforce(self, dataset, direction=1, batch_size=32): # 强化学习训练
    num_minibatches = dataset.data_size // batch_size # 训练批的个数
    for i in range(num_minibatches): # 按批训练
        batch_x, batch_y = dataset.get_batch(batch_size) # 按批获取棋局状态和标签
        self.session.run(self.reinforce_step, feed_dict={self.x: batch_x,
        self.y: batch_y, self.reinforce_direction: direction})
        # 执行训练过程

def run(self, position): # 输入是棋局状态
    processed_position = features.extract_features(position) # 提取棋局特征
    probabilities = self.session.run(self.output, feed_dict={self.
    x: processed_position[None, :]})[0]
    return probabilities.reshape([go.N, go.N]) # 获取所有位置的走子概率

def run_many(self, positions): # 并行计算多个棋局在所有位置的走子概率
    processed_positions = features.bulk_extract_features(positions)
    # 获取多个棋局的特征
    probabilities = self.session.run(self.output, feed_dict={self.
    x: processed_positions})
    return probabilities.reshape([-1, go.N, go.N])

def check_accuracy(self, test_data, batch_size=128): # 检测准确率
    num_minibatches = test_data.data_size // batch_size
    weight_summaries = self.session.run(self.weight_summaries)
    # 获取网络所有层的参数
    for i in range(num_minibatches):
        batch_x, batch_y = test_data.get_batch(batch_size)
        accuracy, cost = self.session.run([self.accuracy, self.log_likelihood_
        cost], feed_dict={self.x: batch_x, self.y: batch_y, self.reinforce_
        direction: 1})
        self.test_stats.report(accuracy, cost)
    avg_accuracy, avg_cost, accuracy_summaries = self.test_stats.collect()
    global_step = self.get_global_step()

```

```

print("Step %s test data accuracy: %g; cost: %g" % (global_step, avg_\
accuracy, avg_cost))
if self.test_summary_writer is not None:    # 保存网络测试过程的结果
    self.test_summary_writer.add_summary(weight_summaries, global_step)
self.test_summary_writer.add_summary(accuracy_summaries, global_step)

```

4. strategies.py 文件的代码实现及说明

(1) MCTSNode 类

```

class MCTSNode():                                # 蒙特卡罗树搜索类
    @staticmethod                                # 指明可以通过类名来访问被 staticmethod 标注的函数
    def root_node(position, move_probabilities): # 创建蒙特卡罗树的根节点
        node=MCTSNode(None,None,0)              # 创建根节点
        node.position=position                   # 初始化根节点的状态
        node.expand(move_probabilities)          # 按一定概率扩展根节点
        return node                             # 返回根节点

    def __init__(self, parent, move, prior):      # 初始化蒙特卡罗树搜索类的变量
        self.parent = parent                    # 指向父节点
        self.move = move                       # 移动搜索位置
        self.prior = prior                     # 当前节点的先验概率
        self.position = None                   # 当前节点的初始状态为空
        self.children = {}                     # 当前节点的初始子节点集合为空
        self.Q = self.parent.Q if self.parent is not None else 0
        # 估值网络的估值
        self.U = prior                         # 用先验概率初始化额外奖励值
        self.N = 0                             # 节点被访问过的次数

    def __repr__(self):                          # 把类对象转换成一个字符串输出
        return "<MCTSNode move=%s prior=%s score=%s is_expanded=%s>" % (self.\
move, self.prior, self.action_score, self.is_expanded())

    @property                                    # 将被注释的方法当作类的一个属性
    def action_score(self):                      # 动作值, 用来选择叶节点进行扩展
        return self.Q + self.U                 # 返回扩展分支的评估值

    def is_expanded(self):                      # 判断当前节点是否被扩展过
        return self.position is not None

    def compute_position(self):                 # 计算走子后的下一个棋局状态
        self.position = self.parent.position.play_move(self.move)
        return self.position

    def expand(self, move_probabilities):        # 扩展叶节点
        self.children = {move: MCTSNode(self,move,prob)
                           for move,prob in np.ndenumerate(move_probabilities)}
        # 根据走子概率模拟节点扩展

        self.children[None] = MCTSNode(self, None, 0) # 不走是合法的

    def backup_value(self, value):              # 回溯更新
        self.N += 1                            # 节点被访问过的次数自加
        if self.parent is None:                 # 不更新父节点的 Q 值和 U 值
            return
        self.Q, self.U = (self.Q + (value-self.Q)/self.N, c_PUCT * math.sqrt\
(self.parent.N) * self.prior / self.N,)
        self.parent.backup_value(-value)

```

```

# 回溯更新时, 己方增强, 则对手减弱, 但绝对值相等
def select_leaf(self):
    current=self
    while current.is_expanded():
        # 判断当前节点是否被扩展过, 记录得分最高的叶节点
        current=max(current.children.values(),key=lambda node: node.\
            action_score)
    return current

```

(2) MCTSPlayerMixin 类

```

class MCTSPlayerMixin:
    # 模拟走子类
    def __init__(self, policy_network, seconds_per_move=5):
        # 初始化函数
        self.policy_network=policy_network
        self.seconds_per_move=seconds_per_move
        self.max_rollout_depth=go.N*go.N*3
        super().__init__()
    def suggest_move(self, position):
        start = time.time()
        move_probs=self.policy_network.run(position)
        # 获取当前棋局在每个位置的走子概率
        root=MCTSNode.root_node(position, move_probs)
        # 以当前棋局为根节点按概率走子扩展
        while time.time()-start < self.seconds_per_move:
            self.tree_search(root)
            print("Searched for %s seconds" % (time.time() - start), file=sys.\
                stderr)
        sorted_moves=sorted(root.children.keys(), key=lambda move, root=root:\
            root.children[move].N, reverse=True)
        for move in sorted_moves:
            if is_move_reasonable(position, move):
                return move
        return None
    def tree_search(self, root):
        print("tree search", file=sys.stderr)
        chosen_leaf=root.select_leaf()
        position=chosen_leaf.compute_position()
        if position is None:
            print("illegal move!", file=sys.stderr)
            del chosen_leaf.parent.children[chosen_leaf.move]
            return
        print("Investigating following position:\n%s" % (chosen_leaf.\
            position,), file=sys.stderr)
        move_probs=self.policy_network.run(position)
        chosen_leaf.expand(move_probs)
        value=self.estimate_value(root, chosen_leaf)
        print("value: %s" % value, file=sys.stderr)
        chosen_leaf.backup_value(value)
        sys.stderr.flush()
    def estimate_value(self, root, chosen_leaf):
        # 综合评估所选叶节点
        # Estimate value of position using rollout only (for now).

```



```

# 评估被拓展节点的状态值方法: 快速模拟 (随机)。综合评价为 (估值网络的估值 + 快速模拟值) / 2
# (TODO: Value network; average the value estimations from rollout + value
# network)
leaf_position = chosen_leaf.position          # 获取所选叶节点的状态
current = copy.deepcopy(leaf_position)        # 深拷贝一份所选叶节点的状态
simulate_game(self.policy_network, current)   # 使用快速走步策略进行模拟
print(current, file=sys.stderr)
perspective = 1 if leaf_position.to_play == root.position.to_play \
else -1                                       # 获取当前玩家
return current.score() * perspective          # 返回当前棋手的得分

```

5. selfplay.py 文件的代码实现及说明

selfplay.py 用有监督学习策略网络的训练结果初始化强化学习策略网络的参数，通过自弈完成强化学习的过程，其具体内容描述如下。

```

net = policy.PolicyNetwork()                  # 构造一个强化学习策略网络
net.initialize_variables('/Users/brilee/dev/MuGo/saved_models/20170718')
# 初始化网络参数
now = time.time()                            # 获取当前时间戳
positions = [go.Position(to_play=go.BLACK if i%2==0 else go.WHITE) for i in
range(2)] # 创建玩家棋局状态
strategies.simulate_many_games(net, net, positions) # 并行模拟多个对弈棋局
print(time.time() - now)
now = time.time()                            # 获取当前时间戳
def get_winrate(final_positions):             # 计算黑棋的胜率
    black_win = [utils.parse_game_result(pos.result()) == go.BLACK for pos in
final_positions]
    return sum(black_win) / len(black_win)     # 黑棋获胜的平均胜率
def extract_moves(final_positions):            # 获取多次模拟的获取赢家和输家
    winning_moves = []                        # 得分玩家
    losing_moves = []                         # 失分玩家
    for final_position in final_positions:     # 遍历所有模拟棋局
        positions_w_context = utils.take_n(strategies.POLICY_CUTOFF_DEPTH,
sgf_wrapper.replay_position(final_position))
        winner = utils.parse_game_result(final_position.result())
        for pwc in positions_w_context:
            if pwc.position.to_play == winner:
                winning_moves.append(pwc)
            else:
                losing_moves.append(pwc)
    return (load_data_sets.DataSet.from_positions_w_context(winning_moves),
load_data_sets.DataSet.from_positions_w_context(losing_moves))

win_percentage = get_winrate(positions)       # 计算黑棋的胜率
winners, losers = extract_moves(positions)    # 获取多次模拟赢家和输家
print(time.time() - now)
net.reinforce(winners, direction=1)           # 对赢家按梯度上升进行强化学习
net.reinforce(losers, direction=-1)           # 对输家按梯度下降进行强化学习

```

12.4.3 MuGo 的学习训练过程

首先, 从 <https://u-go.net/gamerecords/> 下载棋局 (如图 12.15 所示), 这个站点给我们提供了非常丰富的棋局压缩文件, 从 2001 年开始一直到 2017 年的更新的棋局文件。并且从 2016 年开始几乎每个月都会更新棋局文件, 可以根据需要下载这些压缩文件, 用于围棋的训练集。

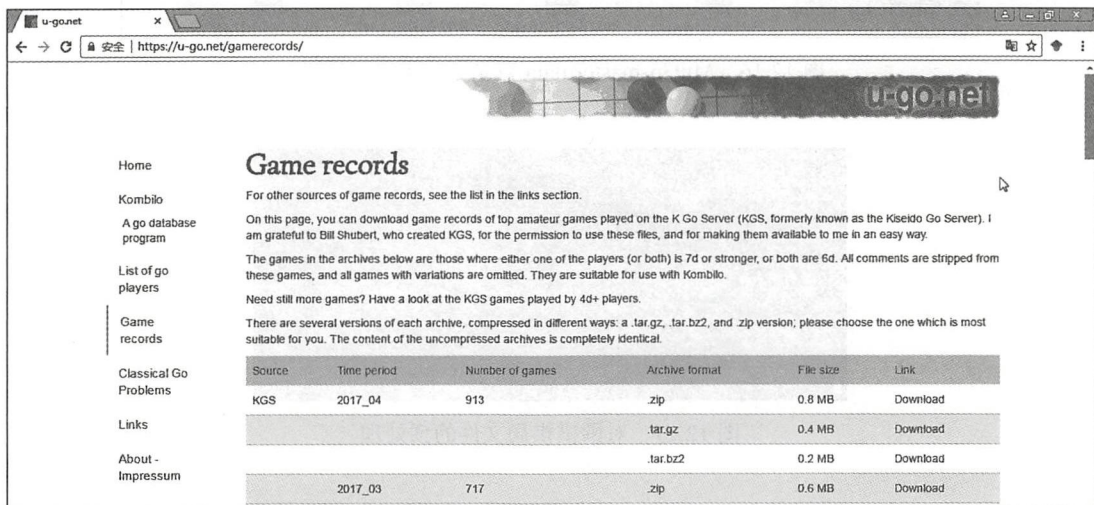


图 12.15 围棋棋局的下载界面

由于硬件资源有限, 虽然下载了几乎所有年份的棋盘压缩文件, 但是在实际训练网络的过程中, 只使用了 2017 年 4 月份及之前年份发布的部分棋局文件, 总共大概包括 22959 个棋局文件。其中 100000 个着子位置用来测试, 其他的用来训练。

下载好这些棋局文件后, 在 MuGo-master 目录下新建一个 data 目录, 用于存放棋局文件, 把下载的棋局文件 *.sgf 解压到 MuGo-master/data 目录下 (如图 12.16 所示)。其次, 对这些棋局文件进行预处理 (如图 12.17 所示), 只需要运行 main.py 文件中的 preprocess 方法就可以对原始棋盘文件进行预处理, 预处理后的数据存放在默认目录 ./processed_data/ (这个目录不需要手动创建, 因为 preprocess 方法已经默认指定把 data 目录下的原始棋盘文件预处理后的文件存放在 processed_data 目录下, 如果 processed_data 目录不存在, 那么程序自动创建这个目录) 下, 预处理阶段比较耗时, 需要耐心等待。预处理完成后就可以使用这些预处理过的文件对 MuGo 的神经网络进行学习和训练, 如图 12.18 和图 12.19 所示。只需要运行 main.py 文件中的 train 方法就可以启动对 MuGo 的神经网络的训练, 训练过程中将网络模型参数保存在 /tmp/ 目录下。最后, 输入命令 “\$ tensorboard --logdir=logs/”, 查看训练反馈结果, 如图 12.20 和图 12.21 所示。

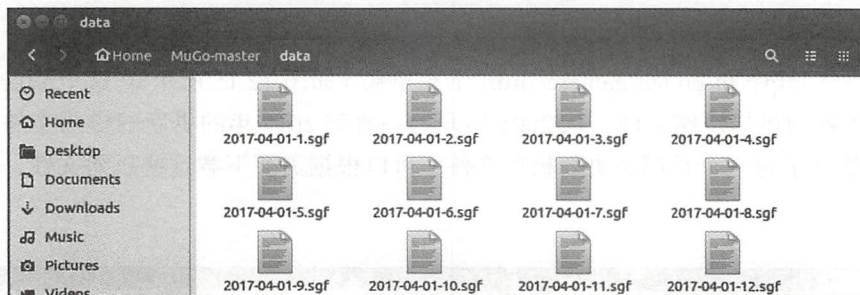


图 12.16 MuGo-master/data 目录下的围棋棋局文件

```
but@ubuntu:~/MuGo-master$ python3 main.py preprocess data/
2017-05-28 01:32:26.380568: W tensorflow/core/platform/cpu_feature_guard.cc:45]
The TensorFlow library wasn't compiled to use SSE4.1 instructions, but these are
available on your machine and could speed up CPU computations.
2017-05-28 01:32:26.381176: W tensorflow/core/platform/cpu_feature_guard.cc:45]
The TensorFlow library wasn't compiled to use SSE4.2 instructions, but these are
available on your machine and could speed up CPU computations.
2017-05-28 01:32:26.381198: W tensorflow/core/platform/cpu_feature_guard.cc:45]
The TensorFlow library wasn't compiled to use AVX instructions, but these are av
ailable on your machine and could speed up CPU computations.
2017-05-28 01:32:26.381210: W tensorflow/core/platform/cpu_feature_guard.cc:45]
The TensorFlow library wasn't compiled to use FMA instructions, but these are av
ailable on your machine and could speed up CPU computations.
22959 sgfs found.
Estimated number of chunks: 1121
Allocating 100000 positions as test; remainder as training
Writing test chunk
```

图 12.17 对围棋棋局文件的预处理

```
but@ubuntu:~/MuGo-master$ python3 main.py train processed_data/ --save-file=/tmp/
2017-05-28 03:45:52.054714: W tensorflow/core/platform/cpu_feature_guard.cc:45] T
he TensorFlow library wasn't compiled to use SSE4.1 instructions, but these are a
vailable on your machine and could speed up CPU computations.
2017-05-28 03:45:52.055131: W tensorflow/core/platform/cpu_feature_guard.cc:45] T
he TensorFlow library wasn't compiled to use SSE4.2 instructions, but these are a
vailable on your machine and could speed up CPU computations.
2017-05-28 03:45:52.055148: W tensorflow/core/platform/cpu_feature_guard.cc:45] T
he TensorFlow library wasn't compiled to use AVX instructions, but these are avai
lable on your machine and could speed up CPU computations.
2017-05-28 03:45:52.055157: W tensorflow/core/platform/cpu_feature_guard.cc:45] T
he TensorFlow library wasn't compiled to use FMA instructions, but these are avai
lable on your machine and could speed up CPU computations.
Using processed data/train746.chunk.gz
Step 128 training data accuracy: 0.00439453; cost: 5.88864
```

图 12.18 MuGo 的学习训练过程

```
but@ubuntu:~/MuGo-master$
Step 1215114 training data accuracy: 0.429932; cost: 2.23849
Saving checkpoint to /tmp/
Using processed data/train745.chunk.gz
Step 1215242 training data accuracy: 0.411377; cost: 2.27075
Saving checkpoint to /tmp/
Using processed data/train731.chunk.gz
Step 1215370 training data accuracy: 0.420898; cost: 2.2672
Saving checkpoint to /tmp/
Using processed data/train713.chunk.gz
Step 1215498 training data accuracy: 0.416992; cost: 2.26125
Saving checkpoint to /tmp/
Using processed data/train927.chunk.gz
Step 1215626 training data accuracy: 0.436279; cost: 2.18856
Saving checkpoint to /tmp/
Using processed data/train320.chunk.gz
Step 1215754 training data accuracy: 0.438477; cost: 2.1609
Saving checkpoint to /tmp/
Using processed data/train291.chunk.gz
Step 1215882 training data accuracy: 0.41333; cost: 2.26838
Saving checkpoint to /tmp/
Using processed data/train752.chunk.gz
Step 1216010 training data accuracy: 0.443359; cost: 2.19831
Saving checkpoint to /tmp/
but@ubuntu:~/MuGo-master$
```

图 12.19 MuGo 的训练结束情况

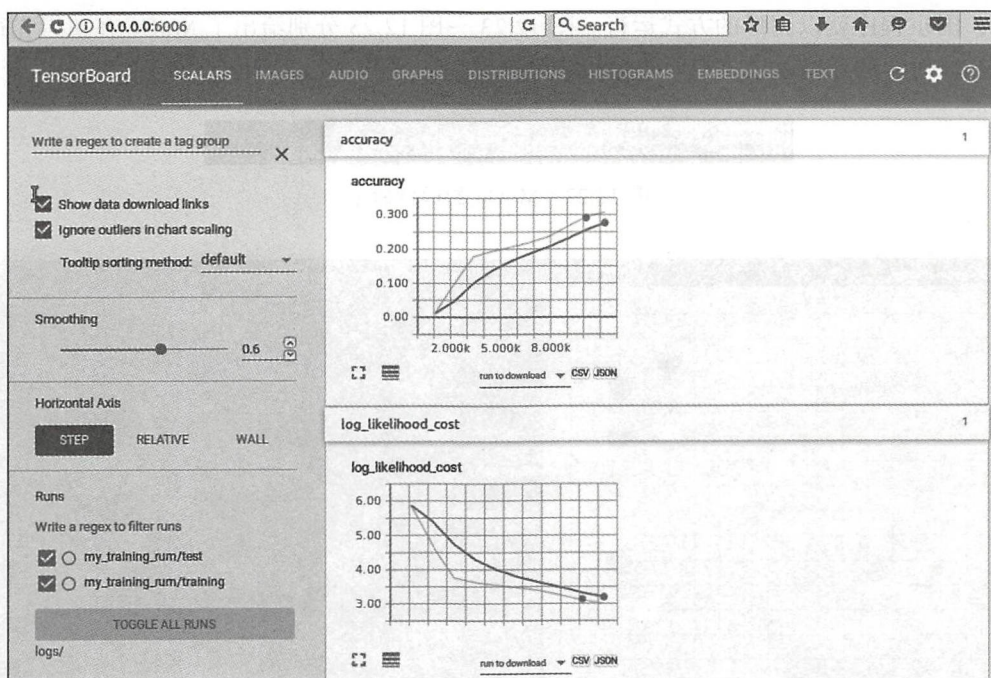


图 12.20 TensorBoard 显示的相关信息

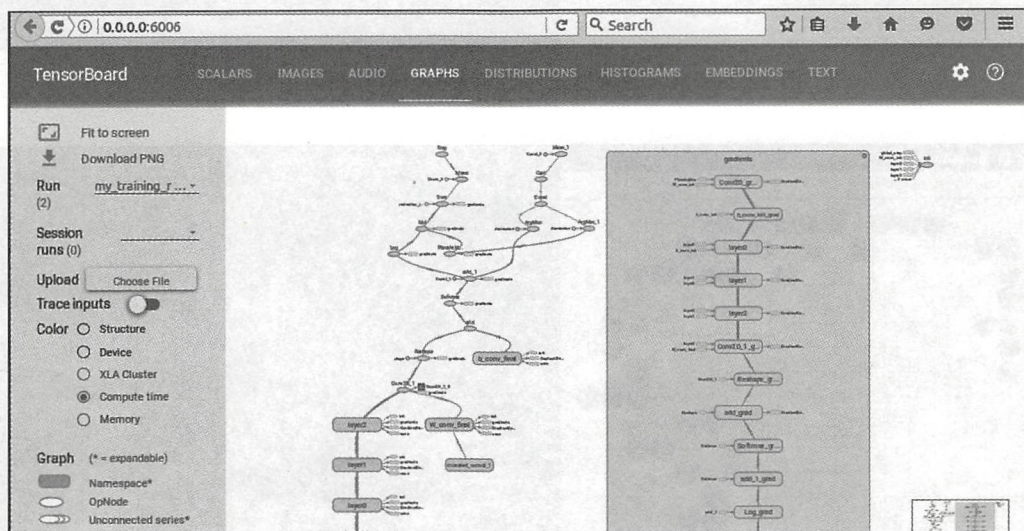


图 12.21 TensorBoard 的 GRAPHS 相关信息

12.4.4 MuGo 的演示效果

在 MuGo 的学习训练完成之后，就可以开始运行了，运行命令如图 12.22 所示。指令执

行后，围棋程序就以图形的方式运行。图 12.23 ~ 图 12.25 分别给出了 MuGo 在执黑先行与业余初段对弈的开局效果、中局效果和结局效果。

```
but@ubuntu:~/MuGo-masters$
but@ubuntu:~/MuGo-masters$ gogul-twogtp -black 'python3 main.py gtp policy --read
-file=/tmp/' -white 'gogul-dtsplay' -size 19 -komi 7.5 -verbose -auto
```

图 12.22 MuGo 的运行命令

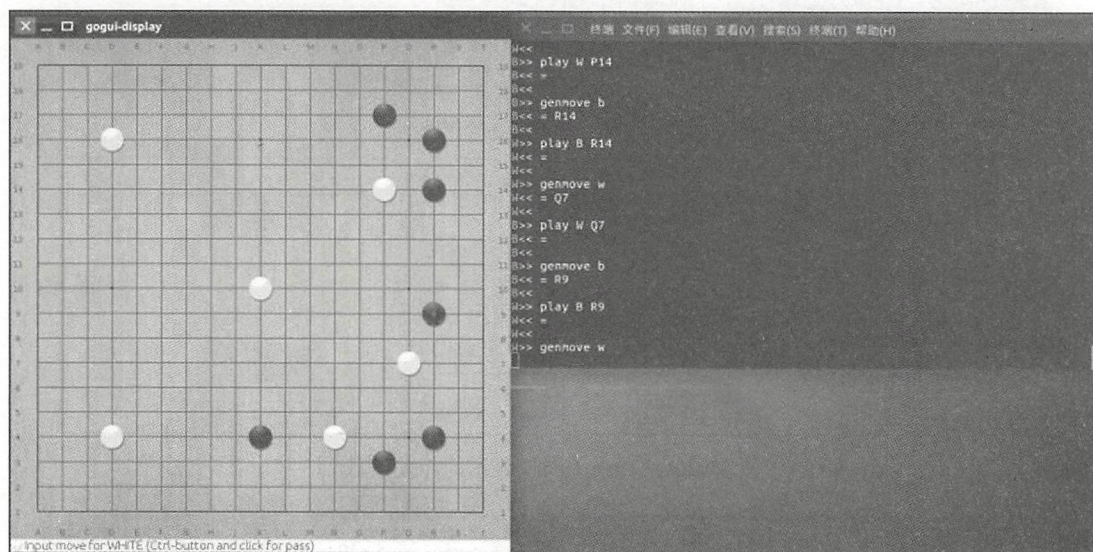


图 12.23 MuGo 执黑先行与业余初段对弈的开局情况

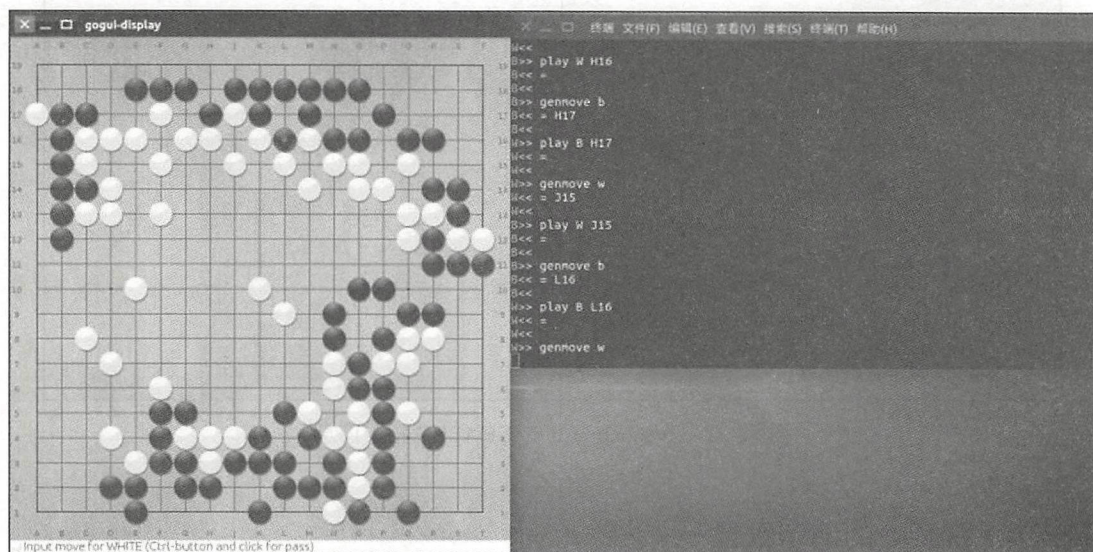


图 12.24 MuGo 执黑先行与业余初段对弈的中局情况

Caffe 在 Windows 上的安装过程

安装环境：Windows 7 64 位操作系统

- 所需软件和库：Visual Studio (VS)、CMake、Anaconda、CUDA、cuDNN、libraries_v140_x64_py27_1.1.0

注意：本书默认，安装 Caffe 之前，需要的相关软件还未安装到计算机上，否则可忽略相关步骤。

1) 下载 Caffe 安装包。从 <https://github.com/BVLC/caffe> 网站上下下载 Caffe 库，将其解压到单独的一个文件夹（如 caffe-windows）中。

2) 安装 VS 2013，或者安装 VS 2015。注意，对于 VS 2015 可能需要先安装 VS 2013，但默认没有安装 C++ 编译器，因此在安装过 VS 2015 之后，还需要通过新建工程安装 C++ 编译器。

3) 安装 CMake。从 <https://cmake.org/download/> 网站下载 CMake 3.8，解压后将 \cmake-3.8.0-win64-x64\bin 路径添加到环境变量中。

4) 安装 Anaconda。从 <https://www.continuum.io/downloads> 网站下载 Anaconda 的 Python2.7（或 Python3.5）版本，然后安装 Anaconda，在安装过程中系统会自动添加环境变量。

5) 安装 CUDA。从 <https://developer.nvidia.com/cuda-downloads> 网站下载 CUDA 8.0，并安装。

6) 安装 cuDNN。从 <https://developer.nvidia.com/cudnn> 网站下载 cuDNN，并安装。

7) 安装依赖库。从 <https://github.com/willyd/caffe-builder/releases> 网站下载 libraries_v140_x64_py27_1.1.0.tar.bz2（或者 libraries_v140_x64_py35_1.1.0），并在 dependencies\libraries_v140_x64_py27_1.1.0 文件夹下解压，如下图所示。

此电脑 > 本地磁盘 (C:) > 用户 > ZhangTing > .caffe > dependencies > libraries_v140_x64_py27_1.1.0			
名称	修改日期	类型	大小
libraries	2017/4/14 11:36	文件夹	
libraries_v140_x64_py27_1.1.0.tar.bz2	2017/4/14 10:46	WinRAR 压缩文件	147,177 KB

8) 注释下载依赖库代码。修改 `caffe-windows\cmake\WindowsDownloadPrebuiltDependencies.cmake` 文件, 将第 71 ~ 78 行注释掉, 以防止程序在编译的时候反复自动下载依赖库, 否则可能会出现长久等待的问题。注释的部分如下图所示。

```

69  if(_download_file)
70      message(STATUS "Downloading prebuilt dependencies to ${_download_path}")
71      #file(DOWNLOAD "${DEPENDENCIES_URL}"
72          #           "${_download_path}"
73          #           EXPECTED_HASH SHA1=${DEPENDENCIES_SHA}
74          #           SHOW_PROGRESS
75          #           )
76      #if(EXISTS ${CAFFE_DEPENDENCIES_DIR}/libraries)
77          #   file(REMOVE_RECURSE ${CAFFE_DEPENDENCIES_DIR}/libraries)
78      #endif()
79  endif()

```

9) 修改配置参数。修改 `caffe-windows\scripts\build_win.cmd` 文件, 主要包括 Python 路径、VS 版本、Python 版本、CPU 等参数。例如,

① 修改第 24 行, 设置 Python 2.7 的安装路径, 即

```

24  set PATH=C:\Users\ZhangTing\Anaconda2;C:\Users\ZhangTing\Anaconda2\Scripts;C:\Users\ZhangTing\Anaconda2\Library\bin;%PATH%

```

② 修改第 71 行, 设置 VS 的版本。如果使用 VS 2015, 则修改为

```

71  if NOT DEFINED MSVC_VERSION set MSVC_VERSION=14

```

如果使用的是 VS 2013, 则修改为

```

71  if NOT DEFINED MSVC_VERSION set MSVC_VERSION=12

```

注意, 这里仅支持 VS 2013 和 VS 2015 两个版本。

③ 修改第 73 行, 选择是否使用 NINJA 编译。由于这里不需要使用, 因此修改为

```

73  if NOT DEFINED WITH_NINJA set WITH_NINJA=0

```

④ 修改第 75 行, 选择是否仅在 CPU 下编译。如果是, 则修改为

```

75  if NOT DEFINED CPU_ONLY set CPU_ONLY=1

```

如果需要同时也在 GPU 下编译, 则修改为

```

75  if NOT DEFINED CPU_ONLY set CPU_ONLY=0

```

⑤ 修改第 83 行, 选择 Python 版本。如果使用 Python 2.7, 则修改为

```

83  if NOT DEFINED PYTHON_VERSION set PYTHON_VERSION=2

```

如果使用 Python 3.5, 则修改为

```
83 if NOT DEFINED PYTHON_VERSION set PYTHON_VERSION=3
```

注意，这里仅支持 Python 2.7 和 Python 3.5 两个版本。

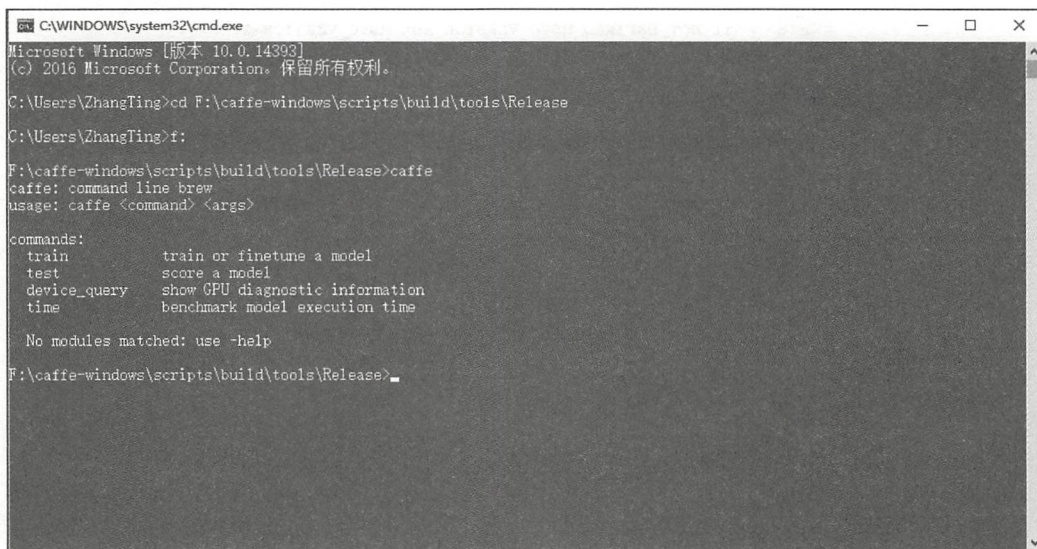
⑥ 如果需要使用 GPU，则还应添加 cuDNN 的路径，可参照下面的示例代码修改第 166 行：

```
155 cmake -G"!CMAKE_GENERATOR!" ^
156     -DBLAS=Open ^
157     -DCMAKE_BUILD_TYPE:STRING=%CMAKE_CONFIG% ^
158     -DBUILD_SHARED_LIBS:BOOL=%CMAKE_BUILD_SHARED_LIBS% ^
159     -DBUILD_python:BOOL=%BUILD_PYTHON% ^
160     -DBUILD_python_layer:BOOL=%BUILD_PYTHON_LAYER% ^
161     -DBUILD_matlab:BOOL=%BUILD_MATLAB% ^
162     -DCPU_ONLY:BOOL=%CPU_ONLY% ^
163     -DCOPY_PREREQUISITES:BOOL=1 ^
164     -DINSTALL_PREREQUISITES:BOOL=1 ^
165     -DUSE_NCCL:BOOL=!USE_NCCL! ^
166     -DCUDNN_ROOT=D:/cudnn_8_v5 ^
167     "%~dp0\..."
```

10) 运行 build。用命令行方式在文件夹 caffe-windows\scripts\ 下运行 build_win.cmd，运行结果存放在 caffe-windows\scripts\build 文件夹下，主要包括 Caffe.sln，以及其他文件和文件夹。

11) 编译 Caffe。用 VS 2015（或 VS 2013）打开 Caffe.sln，依据计算机配置选择 x64 或 x86，并在 Release 下编译，编译结果存放在 \caffe-windows\scripts\build\tools\Release 文件夹下，其中包含 caffe.exe、compute_image_mean.exe、convert_imageset.exe 等可执行文件。

12) 测试是否安装成功。在命令行窗口中进入 caffe.exe 文件所在的目录，输入 caffe，按回车键，如果出现下图类似的情况，则表示安装成功。



```

C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation. 保留所有权利。

C:\Users\ZhangTing>cd F:\caffe-windows\scripts\build\tools\Release
C:\Users\ZhangTing>f:
F:\caffe-windows\scripts\build\tools\Release>caffe
caffe: command line brew
usage: caffe <command> <args>

commands:
  train      train or finetune a model
  test       score a model
  device_query  show GPU diagnostic information
  time       benchmark model execution time

No modules matched: use -help
F:\caffe-windows\scripts\build\tools\Release>

```


Caffe 在 Linux 上的安装过程

安装环境：Ubuntu 16.04 系统

- 所需软件：Anaconda、OpenCV、CUDA、cuDNN。

注意：本书默认，安装 Caffe 之前，需要的相关软件还未安装到计算机上，否则可忽略相关步骤。

1) 安装 Anaconda。从 <https://www.anaconda.com/download/> 网站上下载 Linux Anaconda 4.2 版本。进入 Anaconda3 的下载目录，输入以下命令进行安装：

```
bash Anaconda3-4.2.0-Linux-x86_64.sh
```

注意：安装过程中会遇到 “([y]/n) ?” 的提示，输入 y 即可。

2) 安装依赖包。依次输入以下命令，安装依赖包：

```
sudo apt-get install libprotobuf-dev libleveldb-dev libsnappy-dev libopencv-dev  
libhdf5-serial-dev protobuf-compiler  
sudo apt-get install --no-install-recommends libboost-all-dev  
sudo apt-get install libopenblas-dev liblapack-dev libatlas-base-dev  
sudo apt-get install libgflags-dev libgoogle-glog-dev liblmdb-dev  
sudo apt-get install git cmake build-essential
```

注意：使用以上命令安装依赖包，需要在联网情况下进行。

3) 安装 CUDA。从 <https://developer.nvidia.com/cuda-downloads> 网站上下载 CUDA 8.0 版本后，依次输入以下命令。

```
sudo dpkg -i cuda-repo-ubuntu1604-8-0-local_8.0.44-1_amd64.deb  
sudo apt-get update  
sudo apt-get install cuda
```

安装完成后，需要配置环境变量。首先输入以下命令打开配置文件：

```
sudo gedit ~/.bashrc
```

然后在文件的末尾加入以下两行代码并保存：

```
export PATH=/usr/local/cuda-8.0/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
```

最后输入以下命令使该配置生效：

```
source ~/.bashrc
```

4) 安装 cuDNN。从 <https://developer.nvidia.com/rdp/cudnn-archive> 网站上下载与 CUDA 8.0 版本对应的 cuDNN 的 Linux 系统压缩包，即 cuDNN 5.1。进入存放目录，执行以下命令：

```
tar xvzf cudnn-8.0-linux-x64-v5.1.tgz
sudo cp cudnn.h /usr/local/cuda/include # 复制头文件
sudo cp lib* /usr/local/cuda/lib64/ # 复制动态链接库
cd /usr/local/cuda/lib64/
sudo rm -rf libcudnn.so libcudnn.so.5 # 删除原有动态文件
sudo ln -s libcudnn.so.5.0.5 libcudnn.so.5 # 生成软链接
sudo ln -s libcudnn.so.5 libcudnn.so # 生成软链接
```

5) 安装 OpenCV。从 <https://opencv.org/releases.html> 网站上选择 3.1.0 版本的 Sources，下载 opencv-3.1.0.zip，并将其解压后，进入文件夹 opencv-3.1.0，依次输入以下命令进行编译：

```
mkdir build # 创建编译的文件目录
cd build
cmake -D CMAKE_BUILD_TYPE=Release -D CMAKE_INSTALL_PREFIX=/usr/local ..
make -j8 # 编译
```

最后，输入以下命令进行安装：

```
sudo make install
```

6) 安装 Caffe。在存放路径下输入以下命令下载 Caffe：

```
git clone https://github.com/BVLC/caffe.git
```

然后，将 /caffe 中的 Makefile.config.example 重命名为 Makefile.config，并依次修改里面的配置参数：

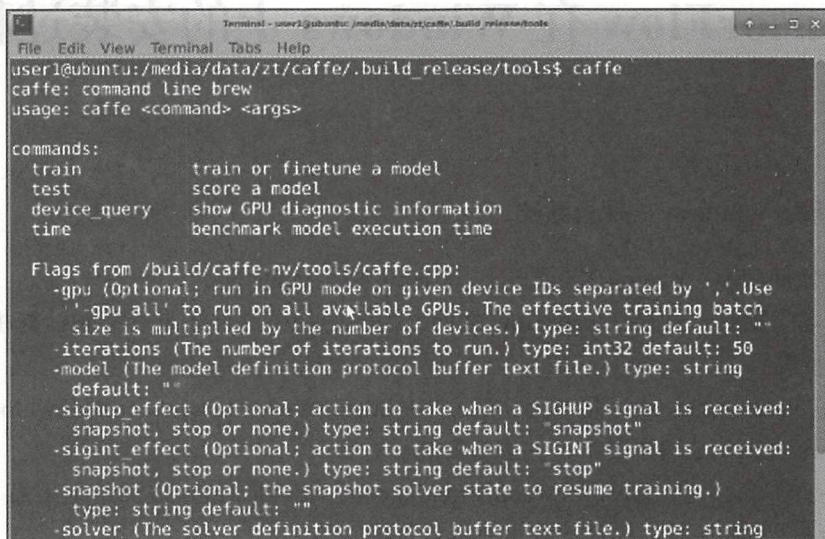
- ① 修改第 5 行，将 # USE_CUDNN:=1 修改为 USE_CUDNN=1，意为使用 cuDNN。
- ② 修改第 21 行，将 #OPENCV_VERSION:=3 修改为 OPENCV_VERSION=3，意为使用 OpenCV3。
- ③ 修改第 73 行，将 #ANACONDA_HOME:=\$(HOME)/anaconda 修改为 ANACONDA_HOME:=\$(HOME)/anaconda3，意为使用 Anaconda3。
- ④ 修改第 92 行，将 #WITH_PYTHON_LAYER:=1 修改为 WITH_PYTHON_LAYER=1，意为使用 Python 接口。

注意：以上操作只是一个示例，如果在实际安装过程中其他参数（例如，使用 Matlab 接口、Python 路径、Python 版本）不同，则需要进行相应的修改。

7) 编译 Caffe。在 `caffe` 目录下输入以下命令，开始编译并安装：

```
make all -j8
```

8) 测试是否安装成功。在 `/caffe/build/tools/` 中，输入 `caffe` 命令，如果出现下图信息，表示安装成功。

A terminal window titled "Terminal - user1@ubuntu: /media/data/zt/caffe/build_release/tools" showing the execution of the 'caffe' command. The prompt is 'user1@ubuntu: /media/data/zt/caffe/.build_release/tools\$'. The output shows 'caffe: command line brew' and 'usage: caffe <command> <args>'. A list of commands is provided: 'train' (train or finetune a model), 'test' (score a model), 'device_query' (show GPU diagnostic information), and 'time' (benchmark model execution time). Below this, flags from the source file are listed, including options for GPU mode, iterations, model definition, and solver state.

```
Terminal - user1@ubuntu: /media/data/zt/caffe/build_release/tools
File Edit View Terminal Tabs Help
user1@ubuntu: /media/data/zt/caffe/.build_release/tools$ caffe
caffe: command line brew
usage: caffe <command> <args>

commands:
  train      train or finetune a model
  test       score a model
  device_query  show GPU diagnostic information
  time       benchmark model execution time

Flags from /build/caffe-nv/tools/caffe.cpp:
-gpu (Optional; run in GPU mode on given device IDs separated by ','. Use
'-gpu all' to run on all available GPUs. The effective training batch
size is multiplied by the number of devices.) type: string default: ""
-iterations (The number of iterations to run.) type: int32 default: 50
-model (The model definition protocol buffer text file.) type: string
default: ""
-sighup_effect (Optional; action to take when a SIGHUP signal is received:
snapshot, stop or none.) type: string default: "snapshot"
-sigint_effect (Optional; action to take when a SIGINT signal is received:
snapshot, stop or none.) type: string default: "stop"
-snapshot (Optional; the snapshot solver state to resume training.)
type: string default: ""
-solver (The solver definition protocol buffer text file.) type: string
```


TensorFlow 在 Windows 上的安装过程

安装环境：Windows 7 64 位系统

- 所需软件和库：Python 3.5 或以上、CUDA8.1、cuDNN。

注意：本书默认，安装 TensorFlow 之前，需要的相关软件还未安装到计算机上，否则可忽略相关步骤。

1) 安装 Python 的集成环境 Anaconda。从 <https://www.anaconda.com/download/> 网站上下载 Windows Anaconda 4.2 版本，并进行安装。

2) 安装 CUDA。从 <https://developer.nvidia.com/cuda-downloads> 网站上下载 CUDA 8.1 版本，分网络版和本地版，推荐下载本地版后直接安装。

3) 下载 cuDNN 库。从 <https://developer.nvidia.com/rdp/cudnn-download> 网站上下载 cuDNN 库，并将其中的 3 个文件夹 bin、lib、include 分别拷贝到 CUDA 的对应文件夹下。

4) 安装 TensorFlow。参考图 11.8 输入命令，默认安装最新版的 TensorFlow。其中，CPU 安装命令为 `pip install tensorflow`，GPU 安装命令为 `pip install tensorflow-gpu`。如需要安装指定版本的 TensorFlow，比如 CPU 1.1.0 版本，则可在命令行窗口输入下面的命令。

```
pip install tensorflow=1.1.0 (或者 pip3 install tensorflow=1.1.0)
```

5) 测试 TensorFlow 是否安装成功。在命令行窗口依次输入以下命令：

```
$ python
...
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()      # 在该步会显示电脑的显卡信息
>>> print(sess.run(hello))
```

如果出现了下图所示的“Hello, TensorFlow!”，则表示安装成功。



TensorFlow 在 Linux 上的安装过程

安装环境：Ubuntu 16.04 系统

- 所需软件和库：Python 3.5 或以上、CUDA8.0、cuDNN

注意：本书默认，安装 TensorFlow 之前，需要的相关软件还未安装到计算机上，否则可忽略相关步骤。

1) 安装 Anaconda。从 <https://www.anaconda.com/download/> 网站上下载 Linux Anaconda 4.2 版本，进入下载目录，输入以下命令进行安装：

```
bash Anaconda3-4.2.0-Linux-x86_64.sh
```

注意：安装过程中会遇到 “([y]/n) ?” 的提示，输入 y 即可。

2) 安装 CUDA。从 <https://developer.nvidia.com/cuda-downloads> 网站上下载 CUDA 8.0 版本，依次输入以下命令。

```
sudo dpkg -i cuda-repo-ubuntu1604-8-0-local_8.0.44-1_amd64.deb
sudo apt-get update
sudo apt-get install cuda
```

安装完成后，还需要配置环境变量。首先用以下命令打开配置文件：

```
sudo gedit ~/.bashrc
```

然后在文件末尾加入下面两行代码并保存：

```
export PATH=/usr/local/cuda-8.0/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
```

最后，输入以下命令使该配置生效：

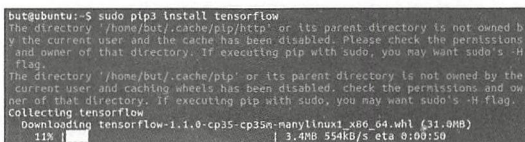
```
source ~/.bashrc
```

3) 安装 cuDNN。从 <https://developer.nvidia.com/rdp/cudnn-archive> 网站上下载对应 CUDA

8.0 的 cuDNN 的 Linux 系统压缩包，即 cuDNN 5.1。进入下载目录，执行以下命令：

```
tar xvzf cudnn-8.0-linux-x64-v5.1.tgz
sudo cp cuda/include/cudnn.h /usr/local/cuda/include
sudo cp cuda/lib64/libcudnn.so* /usr/local/cuda/lib64
sudo chmod a+r /usr/local/cuda/include/cudnn.h /usr/local/cuda/lib64/libcudnn.so*
```

4) 安装 TensorFlow。输入下图所示命令，安装 TensorFlow。

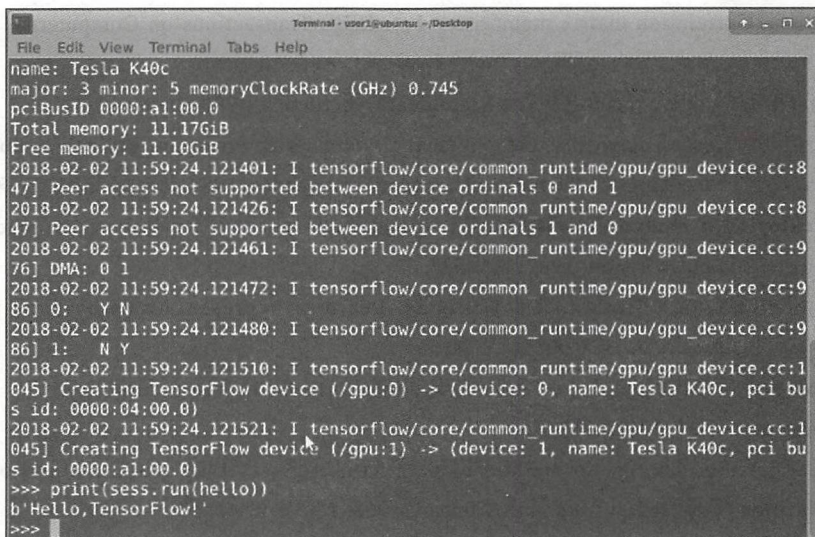


```
but@ubuntu:~$ sudo pip3 install tensorflow
The directory '/home/but/.cache/pip/http' or its parent directory is not owned by
y the current user and the cache has been disabled. Please check the permissions
and owner of that directory. If executing pip with sudo, you may want sudo's -H
flag.
The directory '/home/but/.cache/pip' or its parent directory is not owned by the
current user and caching wheels has been disabled. Check the permissions and ow
ner of that directory. If executing pip with sudo, you may want sudo's -H flag.
Collecting tensorflow
  Downloading tensorflow-1.1.0-cp35-cp35m-manylinux1_x86_64.whl (31.0MB)
    11% | 3.4MB 554kB/s eta 0:00:50
```

5) 测试 TensorFlow 是否安装成功。在命令行中依次输入以下命令：

```
$ python
...
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()          # 在该步会显示电脑的显卡信息
>>> print(sess.run(hello))
```

如果出现了下图所示的“Hello, TensorFlow!”，则表示安装成功。



```
Terminal - user1@ubuntu: ~/Desktop
File Edit View Terminal Tabs Help
name: Tesla K40c
major: 3 minor: 5 memoryClockRate (GHz) 0.745
pciBusID 0000:a1:00:0
Total memory: 11.17GiB
Free memory: 11.10GiB
2018-02-02 11:59:24.121401: I tensorflow/core/common_runtime/gpu/gpu_device.cc:8
47] Peer access not supported between device ordinals 0 and 1
2018-02-02 11:59:24.121426: I tensorflow/core/common_runtime/gpu/gpu_device.cc:8
47] Peer access not supported between device ordinals 1 and 0
2018-02-02 11:59:24.121461: I tensorflow/core/common_runtime/gpu/gpu_device.cc:9
76] DMA: 0 1
2018-02-02 11:59:24.121472: I tensorflow/core/common_runtime/gpu/gpu_device.cc:9
86] 0: Y N
2018-02-02 11:59:24.121480: I tensorflow/core/common_runtime/gpu/gpu_device.cc:9
86] 1: N Y
2018-02-02 11:59:24.121510: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1
045] Creating TensorFlow device (/gpu:0) -> (device: 0, name: Tesla K40c, pci bu
s id: 0000:04:00:0)
2018-02-02 11:59:24.121521: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1
045] Creating TensorFlow device (/gpu:1) -> (device: 1, name: Tesla K40c, pci bu
s id: 0000:a1:00:0)
>>> print(sess.run(hello))
b'Hello, TensorFlow!'
>>>
```

参考文献

- [1] W S McCulloch, W Pitts. A logical calculus of the ideas immanent in nervous activity[J]. The bulletin of Mathematical Biophysics, 1943, 5(4): 115-133.
- [2] D Hebb. The Organization of Behavior: A Neuropsychological Theory[M]. New York: Psychology Press, 1949.
- [3] F Rosenblatt. The Perceptron: A probabilistic model for information storage and organization in the Brain[J]. Psychological Review, 1958, 65(6): 386-408.
- [4] M Minsky, S Papert. Perceptrons: An Introduction to Computational Geometry[M]. Cambridge: The MIT Press, 1969.
- [5] S Grossberg. Some networks that can learn, remember, and reproduce any number of complicated space-time patterns[J]. Journal of Mathematics and Mechanics, 1969, 19(1): 53-91.
- [6] T Kohonen. Correlation matrix memories[J]. IEEE Transactions on Computers, 1972, 100(4): 353-359.
- [7] G Palm. On associative memory[J]. Biological Cybernetics, 1980, 36(1): 19-31.
- [8] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities[J]. Proceedings of the national academy of sciences, 1982, 79(8): 2554-2558.
- [9] D H Ackley, G E Hinton, T J Sejnowski. A learning algorithm for Boltzmann machines[J]. Readings in Computer Vision, 1987: 522-533.
- [10] D E Rumelhart, G E Hinton, R J Williams. Learning representations by back-propagating errors[J]. Nature, 1986, 323(6088): 533-536.
- [11] G Montague, J Morris. Neural-network contributions in biotechnology[J]. Trends in Biotechnology, 1994, 12(8): 312-324.
- [12] A G Ivakhnenko. The group method of data handling – a rival of the method of stochastic approximation[J]. Soviet Automatic Control, 1968, 13(3): 43-45.
- [13] S Haykin, R Lippmann. Neural Networks: A Comprehensive Foundation[J]. International Journal of Neural Systems, 1994, 5(4): 363-364.
- [14] K Fukushima. Neural network model for a mechanism of pattern recognition unaffected by shift in position—Neocognitron[J]. IEICE Technical Report, 1989, 62(10): 658-665.
- [15] K Fukushima. Neocognitron: A self-organizing neural network for a mechanism of pattern

- recognition unaffected by shift in position [J]. Biological Cybernetics, 1980, 36(4): 193-202.
- [16] K Fukushima. Artificial vision by multi-layered neural networks: neocognitron and its advances [J]. Neural Networks, 2013, 37: 103-119.
- [17] D H Hubel, T Wiesel. Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex [J]. Journal of Physiology, 1962, 160(1): 106-154.
- [18] D H Wiesel, T N Hubel. Receptive fields of single neurones in the cat's striate cortex [J]. Journal of Physiology, 1959, 148(3): 574-591.
- [19] Y LeCun, L Bottou, Y Bengio, et al. Gradient-based learning applied to document recognition [J]. Proceedings of IEEE, 1998, 86(11): 2278-2324.
- [20] O Russakovsky, J Deng, H Su, et al. ImageNet large scale visual recognition challenge [J]. Journal of Computer Vision, 2015, 115(3): 211-252.
- [21] P J Werbos. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences [D]. Harvard University, 1974.
- [22] G Cybenko. Approximation by superpositions of a sigmoid function [J]. Mathematics of Control, Signals and Systems, 1989, 2(4): 303-314.
- [23] K Funahashi. On the approximate realization of continuous mappings by neural networks [J]. Neural Networks, 1989, 2(3): 183-192.
- [24] S Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen [D]. Technische Universität München, 1991.
- [25] S Hochreiter, Y Bengio, P Frasconi, J Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies [M]. New York: Wiley, 2001.
- [26] J Schmidhuber. Curious model-building control systems [C]. Proc. IJCNN, 1991: 1458-1463.
- [27] S Hochreiter, F Informatik. Long short-term memory [J]. Neural Computation, 1997, 9(8): 1735-1780.
- [28] J Martens. Deep learning via Hessian-free optimization [C]. Proc. ICML, 2010: 735-742.
- [29] Y LeCun, L Bottou, Y Bengio, P Haffner. Gradient based learning applied to document recognition [J]. IEEE, 1998, 86(11): 2278-2324.
- [30] C Cortes, V Vapnik. Support vector network [J]. Machine Learning, 1995, 20(3): 273-297.
- [31] G E Hinton, S Osindero, Y Teh. A fast learning algorithm for deep belief nets [J]. Neural Computation, 2006, 18(7): 1527-1554.
- [32] G E Hinton, R R Salakhutdinov. Reducing the dimensionality of data with neural networks [J]. Science, 2006, 313(9): 504-507.
- [33] A Fisher, C Igel. Training restricted Boltzmann machines: an introduction [J]. Pattern Recognition, 2014, 47(1): 25-39.
- [34] R Salakhutdinov, G E Hinton. Deep Boltzmann machines [C]. Proc. AIS, 2009: 448-455.
- [35] H Poon, P Domingos. Sum-product networks: A new deep architecture [C]. Proc. ICCV, 2011: 689-697.



- [36] L Deng, X He, J Gao. Deep stacking networks for information retrieval [C]. Proc. ICASSP, 2013: 3153-3157.
- [37] V Mnih, K Kavukcuoglu, D Silver, et al. Human-level control through deep reinforcement learning [J]. Nature, 2015, 518(7540): 529-533.
- [38] I Goodfellow, J Abadie, M Mirza, et al. Generative Adversarial Nets [C]. Proc. 28th NIPS, 2014: 2672-2680.
- [39] J Nagi, F Ducatelle, G A Caro, et al. Max-pooling convolutional neural networks for vision-based hand gesture recognition [C]. Proc. ICSIPA, 2011: 342-347.
- [40] G E Hinton, N Srivastava, A Krizhevsky, et al. Improving neural networks by preventing co-adaptation of feature detectors [J]. Computer Science, 2012, 3(4): 212-223.
- [41] L Wan, M Zeiler, S X Zhang, et al. Regularization of neural networks using dropconnect [C]. Proc. ICML, 2013: 2095-2103.
- [42] A Mohamed, G E Dahl, G E Hinton. Acoustic modeling using deep belief networks [J]. IEEE Transactions on Audio, Speech, and Language Processing, 2012, 20(1): 14-22.
- [43] Y Bengio. Learning deep architectures for AI [J]. Foundation and Trends in Machine Learning, 2009, 2(2): 1-127.
- [44] J Hastad, M Goldmann. On the power of small-depth threshold circuits [J]. Computational Complexity, 1991, 1(2): 113-129.
- [45] Y LeCun, B Boser, J S Denker, et al. Backpropagation applied to handwritten zip code recognition [J]. Neural Computation, 1989, 1(4): 541-551.
- [46] A Krizhevsky, I Sutshever, G E Hinton. ImageNet classification with deep convolutional neural networks [C]. Proc. NIPS, 2012: 4-13.
- [47] A Stuhlsatz, J Lippel, T Zielke. Feature extraction with deep neural networks by a generalized discriminant analysis [J]. IEEE Transactions on Neural Networks and Learning Systems, 2012, 23(4): 596-608.
- [48] S Ji, W Xu, M Yang, et al. 3D convolutional neural networks for human action recognition [J]. IEEE Transactions Pattern Analysis Machine Intelligence, 2013, 35(1): 221-231.
- [49] A L Maas, A Y Hannun, A Y Ng. Rectifier nonlinearities improve neural network acoustic models [C]. Proc. ICML, 2013: 723-729.
- [50] K M He, X Y Zhang, S Q Ren, et al. Delving deep into rectifiers: surpassing human-level performance on ImageNet classification [C]. Proc. ICCV, 2015: 1026-1034.
- [51] D Clevert, T Unterthiner, S Hochreiter. Fast and accurate deep network learning by exponential linear units (ELUs) [C]. Proc. ICLR, 2016: 256-265.
- [52] S Chopra, R Hadsell, Y Lecun. Learning a similarity metric discriminatively, with application to face verification [C]. Proc. CVPR, 2005: 539-546.
- [53] R K Srivastava, K Greff, J Schmidhuber. Highway networks [J]. arXiv:1505.00387, 2015.
- [54] K Simonyan, A Zisserman. Very deep convolutional networks for large-scale image recognition [J].



- arXiv: 1409. 1556, 2014.
- [55] K M He, X Y Zhang, S Q Ren, et al. Delving deep into rectifiers: surpassing human-level performance on ImageNet classification [C]. Proc. ICCV, 2015: 1026-1034.
 - [56] C Szegedy, W Liu, Y Jia, et al. Going deeper with convolutions [C]. Proc. CVPR, 2015: 1-9.
 - [57] M Lin, Q Chen, S Yan. Network in network [J]. arXiv: 1312.4400, 2013.
 - [58] R Girshick, J Donahue, T Darrell, et al. Rich feature hierarchies for accurate object detection and semantic segmentation [C]. Proc. CVPR, 2014: 580-587.
 - [59] K He, X Zhang, S Ren, et al. Spatial pyramid pooling in deep convolutional networks for visual recognition [J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2015, 37(9): 1904-1916.
 - [60] R Girshick. Fast R-CNN [C]. Proc. ICCV, 2015: 1385-1394.
 - [61] S Ren, K He, R Girshick, et al. Faster R-CNN: towards real-time object detection with region proposal networks [C]. Proc. NIPS, 2015: 1-9.
 - [62] J Redmon, S Divvala, R Girshick, et al. You only look once: unified, real-time object detection [C]. Proc. CVPR, 2016: 779-788.
 - [63] W Liu, D Anguelov, D Erhan, et al. SSD: single shot multibox detector [C]. Proc. ECCV, 2016: 21-37.
 - [64] M Jaderberg, K Simonyan, A Zisserman, et al. Spatial transfer networks [C]. Proc. NISP, 2015: 2017-2025.
 - [65] K He, G Gkioxari, P Dollár, et al. Mask r-cnn [C]. Proc. ICCV, 2017: 2980-2988.
 - [66] N Srivastava, G E Hinton, A Krizhevsky, et al. Dropout: a simple way to prevent neural networks from overfitting [J]. Journal of Machine Learning Research, 2014, 15(1): 1929-1958.
 - [67] S Ioffe, C Szegedy, Batch normalization: accelerating deep network training by reducing internal covariate shift [C]. Proc. ICML, 2015: 448-456.
 - [68] K He, X Zhang, S Ren, et al. Deep residual learning for image recognition [C]. Proc. CVPR, 2016: 770-778.
 - [69] F N Iandola, S Han, M W Moskewicz, et al. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 MB model size [J]. arXiv:1602.07360, 2016.
 - [70] S Han, H Mao, W J Dally. Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding [C]. Proc. ICLR, 2016.
 - [71] M D Zeiler, R Fergus. Visualizing and understanding convolutional networks [J]. Lecture Notes in Computer Science, 2013, 8689: 818-833.
 - [72] ILSVRC 2016 [OL]. <http://image-net.org/challenges/LSVRC/2016/results>.
 - [73] Y Chen, J Li, H Xiao, et al. Dual path networks [C]. Proc. NIPS, 2017: 4470-4478.
 - [74] P Sermanet, D Eigen, X Zhang, et al. Overfeat: Integrated recognition, localization and detection using convolutional networks [J]. arXiv: 1312.6229, 2013.
 - [75] X Zeng, W Ouyang, B Yang, et al. Gated bi-directional CNN for object detection [C]. Proc. ECCV,



2016: 354-369.

- [76] H Fan, Z Cao, Y Jiang, et al. Learning deep face representation [J]. arXiv:1403.2802, 2014.
- [77] C Ding, D Tao. Robust face recognition via multimodal deep face representation [J]. IEEE Transactions on Multimedia, 2015, 17(11): 2049-2058.
- [78] Y Sun, D Liang, X Wang, X Tang. DeepID3: face recognition with very deep neural networks [C]. Proc. CVPR, 2015: 356-363.
- [79] F Schroff, D Kalenichenko, J Philbin. FaceNet: a unified embedding for face recognition and clustering [C]. Proc. CVPR, 2015: 24-32.
- [80] D Ciresan, U Meier, J Masci, et al. A committee of neural networks for traffic sign classification [C]. Proc. IJCNN, 2011: 1918-1921.
- [81] D Ciresan, U Meier, J Masci, J Schmidhuber. Multi-column deep neural network for traffic sign classification [J]. Neural Networks, 2012, 32(1): 333-338.
- [82] Q V Le, W Y Zou, S Y Yeung, et al. Learning hierarchical invariant spatio-temporal features for action recognition with independent subspace analysis [C]. Proc. CVPR, 2011: 3361-3368.
- [83] A Karpathy, G Toderici, S Shetty, et al. Large-scale video classification with convolutional neural networks [C]. Proc. CVPR, 2014: 1725-1732.
- [84] J Y H Ng, M Hausknecht, S Vijayanarasimhan, et al. Beyond short snippets: deep networks for video classification [C]. Proc. CVPR, 2015: 4694-4702.
- [85] O Abdel-Hamid, A R Mohamed, H Jiang, et al. Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition [C]. Proc. ICASSP, 2012: 4277-4280.
- [86] O Abdel-Hamid, L Deng, D Yu. Exploring convolutional neural network structures and optimization techniques for speech recognition [J]. Interspeech, 2013, 2013: 1173-1175.
- [87] O Abdel-Hamid, A R Mohamed, H Jiang, et al. Convolutional neural networks for speech recognition [J]. IEEE Transactions on Audio, Speech and Language Processing, 2014, 22(10): 1533-1545.
- [88] L Deng, O Abdel-Hamid, D Yu. A deep convolutional neural network using heterogeneous pooling for trading acoustic invariance with phonetic confusion [C]. Proc. ICASSP, 2013: 6669-6673.
- [89] T N Sainath, B Kingsbury, A R Mohammed, et al. Learning filter banks within a deep neural network framework [C]. Proc. ASRU, 2013: 297-302.
- [90] T N Sainath, A R Mohamed, B Kingsbury, et al. Deep convolutional neural networks for LVCSR [C]. Proc. ICASSP, 2013: 8614-8618.
- [91] J Gehring, M Auli, D Grangier, et al. Convolutional sequence to sequence learning [J]. arXiv: 1705.03122v1, 2017.
- [92] Y Wu, M Schuster, Z Chen, et al. Google ' s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation [J]. arXiv:1609.08144, 2016.
- [93] D Silver, A Huang, C J Maddison, et al. Mastering the game of Go with deep neural networks and tree search [J]. Nature, 2016, 529(7587): 484-489.



- [94] B M Lake, T D Ullman, J B Tenenbaum. Building machines that learn and think like people[J]. Behavioral and Brain Sciences, 2016, 40:1-58.
- [95] I J Goodfellow, J Shlens, C Szegedy. Explaining and harnessing adversarial examples[C]. Proc. ICLR, 2015: 1-12.
- [96] A Nguyen, J Yosinski, J Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images[C]. Proc. CVPR, 2015: 427-436.
- [97] A Mahendran, A Vedaldi. Visualizing deep convolutional neural networks using natural pre-images[J]. International Journal of Computer Vision, 2016, 120(3): 233-255.
- [98] V Badrinarayanan, B Mishra, R Cipolla. Understanding neural networks through deep visualization[C]. Proc. ICML, 2015: 1-10.
- [99] A Krizhevsky. One weird trick for parallelizing convolutional neural networks[J]. arXiv: 1404.5997, 2014.
- [100] 孙文瑜, 徐成贤, 朱德通. 最优化方法[M]. 北京: 高等教育出版社, 2005.
- [101] X Glorot, Y Bengio. Understanding the difficulty of training deep feedforward neural networks[C]. Proc. ICAIS, 2010: 249-256.
- [102] K He, X Zhang, S Ren, et al. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification[C]. Proc. ICCV, 2015: 1026-1034.
- [103] Y Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$ [C]. Doklady Akad. Nauk SSSR, 1983, 269: 543-547.
- [104] J Duchi, E Hazan, Y Singer. Adaptive subgradient methods for online learning and stochastic optimization[J]. Journal of Machine Learning Research, 2011, 12(Jul): 2121-2159.
- [105] M D Zeiler. ADADELTA: an adaptive learning rate method[J]. arXiv:1212.5701, 2012.
- [106] M Riedmiller, H Braun. RPROP-A fast adaptive learning algorithm[C]. Proc. ISCIS, 1992.
- [107] D P Kingma, J Ba. Adam: A method for stochastic optimization[C]. Proc. ICLR, 2015.
- [108] R Bellman. Dynamic programming[J]. Courier Corporation, 2013.
- [109] A Krizhevsky, I Sutskever, G E Hinton. Imagenet classification with deep convolutional neural networks[J]. Advances in neural information processing systems. 2012: 1097-1105.
- [110] K Grauman, T Darrell. The pyramid match kernel: Discriminative classification with sets of image features[C]. Proc. ICCV, 2005: 1458-1265.
- [111] S Lazebnik, C Schmid, J Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories[C]. Proc. CVPR, 2006: 2169-2178.
- [112] K He, X Zhang, S Ren, J Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2014, 37(9): 1904-1916.
- [113] J Yang, K Yu, Y Gong, T Huang. Linear spatial pyramid matching using sparse coding for image classification[C]. Proc. CVPR, 2009: 1794-1801.
- [114] J Wang, J Yang, K Yu, F Lv, T Huang, Y Gong. Locality constrained linear coding for image



- classification [C]. Proc. CVPR, 2010: 3360-3367.
- [115] K E van de Sande, J R Uijlings, T Gevers, A W Smeulders. Segmentation as selective search for object recognition [C]. Proc. ICCV, 2012: 1879-1886.
- [116] Y Jia. Caffe: An open source convolutional architecture for fast feature embedding [OL]. <http://caffe.berkeleyvision.org/>, 2013.
- [117] M D Zeiler, R Fergus. Visualizing and understanding convolutional neural networks [J]. arXiv:1311.2901, 2013.
- [118] P Sermanet, D Eigen, X Zhang, M Mathieu, R Fergus, Y LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks [J]. arXiv:1312.6229, 2013.
- [119] L Fei-Fei, R Fergus, P Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories [C]. Proc. CVPR, 2007: 178-187.
- [120] M Everingham, L Van Gool, C K I Williams, J Winn, A Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results [OL]. <http://host.robots.ox.ac.uk/pascal/VOC/voc2007/results/index.shtml>.
- [121] K Simonyan, A Zisserman. Very deep convolutional networks for large-scale image recognition [J]. arXiv:1409.1556, 2014.
- [122] S Arora, A Bhaskara, R Ge, et al. Provable bounds for learning some deep representations [C]. Proc. ICML, 2014: 584-592.
- [123] C Szegedy C, W Liu, Y Jia, et al. Going deeper with convolutions [C]. Proc. CVPR, 2015: 1-9.
- [124] S Ioffe, C Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift [C]. Proc. ICML, 2015: 448-456.
- [125] C Szegedy, V Vanhoucke, S Ioffe, et al. Rethinking the inception architecture for computer vision [C]. Proc. CVPR, 2016: 2818-2826.
- [126] C Szegedy, S Ioffe, V Vanhoucke, et al. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning [C]. Proc. AAAI. 2017: 4278-4284.
- [127] R K Srivastava, K Greff, J Schmidhuber. Highway networks [J]. arXiv:1505.00387, 2015.
- [128] K He, J Sun. Convolutional neural networks at constrained time cost [C]. Proc. CVPR, 2015: 5353-5360.
- [129] S Xie, R Girshick, P Dollar, et al. Aggregated residual transformations for deep neural networks [C]. Proc. CVPR, 2017: 5987-5995.
- [130] T Y Lin, P Dollar, R Girshick, et al. Feature pyramid networks for object detection [C]. Proc. CVPR, 2017: 2117-2125.
- [131] S Zagoruyko, N Komodakis. Wide residual networks [J]. arXiv:1605.07146, 2016.
- [132] G Huang, Z Liu, K Q Weinberger, et al. Densely connected convolutional networks [C]. Proc. CVPR, 2017: 4700-4708.
- [133] T Zhang, Y Li, Z Liu. Shortcut convolutional neural networks for classification of gender and texture [C]. Proc. ICANN, 2017: 30-39.

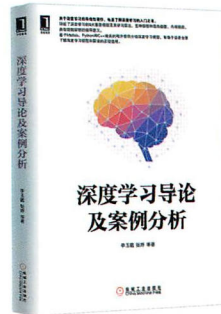


- [134] R Girshick, J Donahue, T Darrell, et al. Rich feature hierarchies for accurate object detection and semantic segmentation[C]. Proc. CVPR, 2014: 580-587.
- [135] J Uijlings, K van de Sande, T Gevers, A Smeulders. Selective search for object recognition[C]. Proc. IJCV, 2013: 154-171.
- [136] B Alexe, T Deselaers, V Ferrari. Measuring the objectness of image windows[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2012: 2189-2202.
- [137] I Endres, D Hoiem. Category independent object proposals[C]. Proc. ECCV, 2010.
- [138] J Carreira, C Sminchisescu. CPMC: Automatic object segmentation using constrained parametric min-cuts[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2012: 1312-1328.
- [139] P Felzenszwalb, R Girshick, D McAllester, D Ramanan. Object detection with discriminatively trained part based models[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2010: 1627-1645.
- [140] P Sermanet, D Eigen, X Zhang, et al. Overfeat: Integrated recognition, localization and detection using convolutional networks[J]. arXiv:1312.6229, 2013.
- [141] R Girshick. Fast R-CNN[C]. Proc. ICCV, 2015: 1440-1448.
- [142] S Ren, K He, R Girshick, et al. Faster r-cnn: Towards real-time object detection with region proposal networks[C]. Proc. NIPS, 2015: 91-99.
- [143] J K Chorowski, D Bahdanau, D Serdyuk, et al. Attention-based models for speech recognition[C]. Proc. NIPS, 2015: 577-585.
- [144] J Long, E Shelhamer, T Darrell. Fully convolutional networks for semantic segmentation[C]. Proc. CVPR, 2015.
- [145] J Redmon, S Divvala, R Girshick, et al. You only look once: Unified, real-time object detection[C]. Proc. CVPR, 2016: 779-788.
- [146] J Redmon, A Farhadi. YOLO9000: Better, Faster, Stronger[C]. Proc. CVPR, 2017: 6517-6525.
- [147] J R Uijlings, K E van de Sande, T Gevers, A W Smeulders. Selective search for object recognition[C]. Proc. IJCV, 2013.
- [148] W Liu, D Anguelov, D Erhan, et al. Ssd: Single shot multibox detector[C]. Proc. ECCV, 2016: 21-37.
- [149] A G Howard. Some improvements on deep convolutional neural network based image classification[J]. arXiv: 1312.5402, 2013.
- [150] J Long, E Shelhamer, T Darrell. Fully convolutional networks for semantic segmentation[C]. Proc. CVPR, 2015: 3431-3440.
- [151] H Zhao, J Shi, X Qi, et al. Pyramid scene parsing network[J]. arXiv:1612.01105, 2016.
- [152] K He, G Gkioxari, P Dollár, et al. Mask r-cnn[C]. Proc. ICCV, 2017: 2980-2988.
- [153] T Y Lin, P Dollar, R Girshick, et al. Feature pyramid networks for object detection[C]. Proc. CVPR, 2017: 2117-2125.
- [154] S Chopra, R Hadsell, Y LeCun. Learning a similarity metric discriminatively, with application to

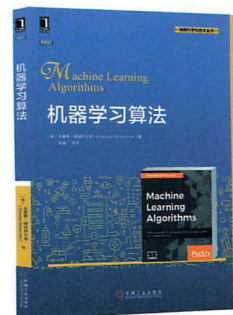


- face verification [C]. Proc. CVPR, 2005: 539-546.
- [155] Y Sun, X Wang, X Tang. Deep learning face representation from predicting 10,000 classes [C]. Proc. CVPR, 2014: 1891-1898.
- [156] Y Sun, X Wang, X Tang. Deeply learned face representations are sparse, selective, and robust [C]. Proc. CVPR, 2015: 2892-2900.
- [157] Y Sun, D Liang, X Wang, et al. Deepid3: Face recognition with very deep neural networks [J]. arXiv:1502.00873, 2015.
- [158] F N Iandola, S Han, M W Moskewicz, et al. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 MB model size [J]. arXiv:1602.07360, 2016.
- [159] A Radford, L Metz, S Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks [J]. arXiv:1511.06434, 2015.
- [160] I Goodfellow, J Pouget-Abadie, M Mirza, et al. Generative adversarial nets [C]. Proc. NIPS, 2014: 2672-2680.
- [161] M Lin, Q Chen, S Yan. Network in network [J]. arXiv:1312.4400, 2013.
- [162] R S Sutton, A G Barto. Reinforcement learning: an introduction [M]. Cambridge: MIT press, 1998.
- [163] V Mnih, K Kavukcuoglu, D Silver, et al. Human-level control through deep reinforcement learning [J]. Nature, 2015, 518(7540): 529-541.
- [164] H V Van, A Guez, D Silver. Deep reinforcement learning with double Q-learning [C]. Proc. AAAI, 2016: 2094-2100.
- [165] M Hausknecht, P Stone. Deep recurrent Q-learning for partially observable MDPs [C]. Proc. SDMIZ, 2015: 29-37.
- [166] Z Wang, N D Freitas, M Lanctot. Dueling network architectures for deep reinforcement learning [C]. Proc. ICML, 2016: 1995-2003.
- [167] T D Kulkarni, K R Narasimhan, A Saeedi, et al. Hierarchical deep reinforcement learning: integrating temporal abstraction and intrinsic motivation [C]. Proc. NIPS, 2016: 3675-3683.
- [168] J Oh, V Chockalingam, S Singh, et al. Control of memory, active perception, and action in Minecraft [C]. Proc. ICML, 2016: 2790-2799.

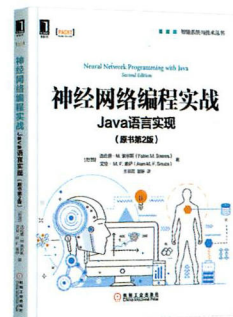
推荐阅读



书号: 978-7-111-55075-4
定价: 59.00元



书号: 978-7-111-59513-7
定价: 69.00元



书号: 978-7-111-60012-1
定价: 59.00元

Deep Learning

Mastering Convolutional Neural Networks from Beginner

卷积神经网络是深度学习应用最广的模型，几乎成了深度学习的代名词。它特别适合图像分类和识别、目标的分割和检测以及人工智能游戏领域的应用。

本书旨在全面介绍各种卷积神经网络的模型、算法及应用，指导读者把握其形成和演变的基本脉络，以帮助读者在较短的时间内从入门达到精通的水平。有兴趣的读者可以从本书开始，通过图像分类、识别、检测和分割的案例，逐步深入卷积神经网络的核心，掌握深度学习的方法和精髓，领会AlphaGo战胜人类世界冠军的奥秘。

本书特色

- 对卷积神经网络进行由浅入深的分类描述，从现代雏形、突破模型、加深模型一直到强化模型和顶尖成就，并附有相关参考文献，方便读者及时查阅和钻研。
- 对卷积神经网络模型的涵盖非常全面，详细讨论了LeNet、AlexNet、VGGNet、GoogLeNet、SPPNet、ResNet、DenseNet、Faster R-CNN、YOLO、SSD、FCN、PSPNet、Mask R-CNN、SiameaseNet、SqueezeNet、DCGAN、NIN和DQN等模型。
- 结合Caffe或TensorFlow代码，详细分析大量卷积神经网络的应用案例，包括字符识别、交通标志识别、交通路网提取、大规模图像分类、人脸图像性别分类、图像目标检测、图像语义分割、图像实例分割、人脸图像生成、Flappy Bird智能体等。
- 通过应用案例的分析，向读者介绍卷积神经网络的配置技巧和操作经验，包括数据增强、图像预处理、网络初始化、激活函数选择，以及训练测试细节等。
- 通过阐述人工智能围棋程序AlphaGo及AlphaGo Zero的设计原理和设计思想，专门探讨AlphaGo的仿效围棋程序——MuGo，以帮助读者自己动手实现一个类似程序。

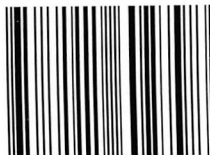


投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/人工智能

ISBN 978-7-111-60279-8



9 787111 602798 >

定价: 79.00元